ANA FLÁVIA BARRETO DE GODOI

# UMA FERRAMENTA PARA COMUNICAÇÃO CONFIÁVEL EM SISTEMAS P2P BASEADA EM GRUPOS DE PEERS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Elias P. Duarte Jr.

CURITIBA

2007

# SUMÁRIO

# Lista de Figuras

# Resumo

O desenvolvimento de sistemas *Peer-to-Peer* (P2P) trouxe novas possibilidades para a construção de aplicações para a Internet, na medida em que explora o grande poder computacional das máquinas conectadas à rede. Exemplos de aplicações incluem o compartilhamento de conteúdo e recursos de forma descentralizada. Os sistemas P2P são formados por processos espalhados pela rede, que possuem a mesma funcionalidade e realizam as tarefas de maneira descentralizada. Este trabalho apresenta uma ferramenta baseada em comunicação em grupos, que pode ser utilizada para a construção de sistemas P2P confiáveis. O serviço de grupos (*group membership*) foi implementado para fornecer comunicação confiável entre *peers*. Um grupo de *peers* disponibiliza conteúdo ou recursos como se fosse um único *peer*. A falha de um membro do grupo não causa interrupção do serviço. Um algoritmo de eleição de líder foi implementado e permite determinar qual componente do grupo atende a cada requisição. A ferramenta desenvolvida também implementa o protocolo de confirmação em duas fases para garantir atomicidade de um conjunto de ações, permitindo a alteração de conteúdo replicado. Uma aplicação P2P para compartilhamento de arquivos foi construída, utilizando os serviços oferecidos pela ferramenta desenvolvida. Esta aplicação foi implementada na plataforma JXTA. Finalmente, um estudo de caso foi avaliado para determinar a latência da detecção entrada e saída de membros do grupo, para medir o tempo necessário para a entrega de mensagens do protocolo de confirmação e para a eleição de *peers* para atender requisições.

# Abstract

The development of Peer-to-Peer (P2P) systems has allowed the development of novel Internet applications, such as content and resource sharing. P2P systems are composed of network processes spread throughout the network, notably located at the border. These processes have the same functionality, tasks can be executed in a decentralized way. This work presents a tool based on group membership, that can be used to construct dependable P2P systems. The group membership service was implemented to manage the interaction between peers. A group abstraction offers a tranparent interface to the users, in the sense that the group is seen as a single peer. The fault of a member peer does not cause service interruptions if the group still has fault-free members. An election algorithm was implemented and allows the determination of which group member will attend each user request. The developed tool also implements the two-phase commit protocol in order to provide the atomic delivery of a set of messages within the group, allowing content update. A P2P content sharing application was built as an example. This application was implemented using the JXTA P2P development platform. Case studies are presented showing representative values for the latency of several system components, such as fault detection, changes in membership caused by join and leave action, the time required to deliver messages sent with the commit protocol and the election of a member to server user requests.

# Capítulo 1

# Introdução

As tecnologias *Peer-To-Peer* (P2P) têm sido bastante utilizadas para o desenvolvimento de sistemas para a Internet, principalmente devido à descentralização e à capacidade de permitir o compartilhamento de conteúdo e recursos computacionais espalhados pela rede, através da cooperação entre os seus integrantes [57]. O grande desenvolvimento dos sistemas P2P começou com a popularização de sistemas para troca de conteúdo como músicas e vídeos [7, 20, 21, 25, 38, 45, 46], mas também tem aplicação em outras áreas como computação distribuída [1, 23], comunicação e trabalho colaborativo [15].

O interesse pelo desenvolvimento de aplicações P2P se baseia nas propriedades que esta tecnologia pode oferecer, como autonomia para os integrantes do sistema, escalabilidade, disponibilidade, anonimato de usuários e recursos, entre outros. Estas propriedades são consequência das características dos sistemas P2P, cuja principal delas é a descentralização. As atividades e recursos do sistema podem estar espalhados entre os processos que formam o sistema, também chamados de *peers*. Cada *peer* possui a capacidade de servir requisições, ao mesmo tempo em que pode fazer requisições a outros *peers* [44, 53, 57]. Os *peers* podem iniciar ou encerrar sua participação nos sistemas P2P a qualquer momento, tornando o sistema dinâmico no que diz respeito ao conjunto de *peers* executando.

Embora sistemas P2P sejam naturalmente disponíveis e escaláveis, a forma dinâmica e autônoma como os *peers* se estruturam na rede para a oferta dos serviços e informações, impossibilita que os mesmos sejam providos de forma contínua. Desta forma, interrup-

ções totais, bem como, variações bruscas de desempenho na oferta desses serviços, são fenômenos frequentes. De fato, nesses sistemas, após estabelecida a rota para o uso do serviço ou a transferência de conteúdo, fica a cargo do usuário a gestão completa da sua utilização. Para a construção de sistemas P2P robustos, torna-se necessário definir e utilizar técnicas apropriadas, que garantam a sua disponibilidade de forma constante, permitindo que os serviços continuem a ser executados de maneira correta e transparente ao usuário, sem que haja muita degradação no seu desempenho, apesar da ocorrência de falhas.

A replicação é uma técnica frequentemente empregada para o aumento da disponibilidade, já que falhas em algumas das suas réplicas não afetam por completo a oferta do serviço [16, 9, 48, 52]. As estratégias de replicação permitem com que grupos de processos distintos, que mantêm cópias dos dados ou serviços, interajam ao longo do tempo para manter o sistema funcionando adequadamente, de maneira transparente ao usuário. Uma propriedade que deve ser garantida é a consistência entre as informações que cada cópia mantém. A consistência pode ser atingida através do controle da comunicação entre os processos e das ações realizadas pelo sistema ao longo do tempo. Uma das abordagens utilizadas para suportar a replicação de maneira consistente é a utilização de serviços de gestão de grupos [13, 30, 28].

Um grupo agrega um conjunto de entidades que compartilham um estado comum e cooperam entre si para a realização de tarefas. Para os seus usuários, ele representa uma entidade lógica única. Um serviço de gestão de grupos assegura a disponibilidade e o estado consistente da aplicação distribuída baseada no grupo, mesmo com a ocorrência de falhas. Ele contempla dois principais componentes. O serviço de gestão da composição (*group membership*) é responsável por criar e manter um histórico coerente da evolução do grupo, com relação a entradas e saídas de membros ou a falhas de processos ou de canais de comunicação. O serviço de gestão da comunicação (*group communication*) oferece aos usuários do grupo e aos seus membros, protocolos de comunicação que permitem com que haja a evolução consistente do estado da aplicação.

Este trabalho apresenta uma ferramenta que oferece um serviço de gestão de grupos

para sistemas P2P, protocolo de confirmação de mensagens e eleição de líder, implementada na plataforma JXTA [35]. O grupo de *peers* disponibiliza conteúdo ou recursos de maneira transparente ao usuário, com garantia de fornecimento do serviço, mesmo no caso de falha de *peers* do grupo. Um membro do grupo é eleito para atender uma requisição pelo serviço, através de um protocolo de eleição de líder. A falha do membro, enquanto este serve requisições, faz com que o grupo determine um membro sem-falha para realizar as tarefas do *peer* falho. O protocolo de confirmação em duas fases permite alteração de conteúdo replicado e pode ser aplicado a sistemas P2P para edição compartilhada de arquivos. Como exemplo de utilização desta ferramenta, foi desenvolvida uma aplicação P2P para *download* confiável de arquivos, no qual o *peer* cliente obtém o arquivo de forma transparente, mesmo no caso de falha do membro do grupo que está lhe enviando o arquivo.

O restante deste trabalho está organizado da seguinte maneira. O capítulo 2 apresenta os fundamentos dos sistemas P2P, suas características e propriedades, citando exemplos de sistemas ao longo do texto. Este capítulo também traz uma descrição do JXTA, que é uma plataforma utilizada para o desenvolvimento de sistemas P2P. O capítulo 3 descreve alguns dos fundamentos para a construção de sistemas distribuídos confiáveis, como os protocolos de confirmação, focando em estratégias para garantir disponibilidade, como a aplicação de técnicas de replicação e ferramentas de gestão de grupos. O capítulo 4 apresenta a ferramenta desenvolvida, descrevendo sua estrutura e os algoritmos de gestão de grupos, protocolo de confirmação e eleição de líder implementados. O capítulo 5 descreve a aplicação para compartilhamento de arquivos construída a partir dos módulos de gestão de grupos e eleição, detalhando os recursos JXTA utilizados, bem como apresenta os resultados obtidos com o estudo de caso apresentado. Por fim, o capítulo 6 faz a conclusão deste trabalho.

# Capítulo 2

# Sistemas *Peer-To-Peer*

O conceito de sistemas *Peer-To-Peer* (P2P) foi criado ainda nos anos 60, no início do desenvolvimento da ARPANET, uma rede totalmente descentralizada e que originou a atual Internet [26]. Apesar do conceito de sistemas *Peer-To-Peer* ter influenciado o desenvolvimento da ARPANET, a maioria dos serviços disponibilizados hoje através da Internet baseia-se em um paradigma diferente, o cliente-servidor, descrito a seguir.

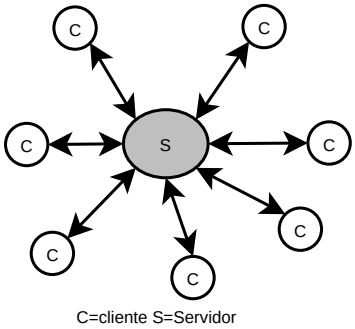


Figura 2.1: Exemplo de um sistema cliente-servidor.

Como ilustrado na figura 2.1, os sistemas cliente-servidor são baseados em processos de dois tipos distintos. Os servidores, que na maioria dos casos executam em máquinas com elevada capacidade computacional, centralizam os serviços e informações disponibilizados na rede. Em contrapartida, os processos clientes, que oferecem interface para o usuário, acessam os serviços através da comunicação com os servidores [14]. Em geral, as máquinas que executam os clientes não precisam ter grande capacidade computacional, uma vez que

a maior parte do processamento das tarefas pode ser feito no servidor.

Entretanto, desde meados dos anos 90 os sistemas baseados na tecnologia P2P têm sido muito utilizados para o desenvolvimento de aplicações para a Internet. Ao contrário dos sistemas cliente-servidor, nos sistemas P2P os processos que se comunicam, chamados *peers* ou nodos, têm basicamente a mesma funcionalidade, mantendo os serviços e informações disponibilizados na rede e oferecendo interface para usuários. Em outras palavras, os *peers* possuem ao mesmo tempo as funcionalidades de clientes e servidores.

Entre as aplicações dos sistemas P2P estão principalmente sistemas para compartilhamento e troca de arquivos, que tiveram seu desenvolvimento impulsionado a partir do surgimento do Napster [45]. Outros exemplos incluem o Kazaa [38], eDonkey [20], o eMule [21], e mais recentemente, o BitTorrent [7]. Há também diversos sistemas para computação distribuída, como o Seti@Home[1], além de outras aplicações que incluem sistemas de comunicação e colaboração [15] e bancos de dados distribuídos [46].

Com a disseminação de tecnologias P2P, também foram desenvolvidas ferramentas ou plataformas para facilitar a implementação das aplicações baseadas nesta tecnologia. Dentre estas ferramentas, há o JXTA (denominação derivada da palavra *juxtapose*) [35], uma biblioteca Java/C disponibilizada como software livre, que provê as funcionalidades necessárias ao desenvolvimento de aplicações P2P, e que é utilizada neste trabalho.

Este capítulo apresenta os sistemas P2P e suas características e mostra suas possíveis organizações e aplicações. Em seguida é descrito o sistema JXTA, suas características, protocolos e funcionalidades.

## 2.1 *Definição de Sistemas Peer-To-Peer*

A tecnologia *Peer-To-Peer* (P2P) têm sido muito utilizada para o desenvolvimento de sistemas para a Internet, devido principalmente à sua característica de permitir o compartilhamento de recursos computacionais espalhados pela rede, através da cooperação entre os seus integrantes. A característica principal destes sistemas é a descentralização do processamento, do armazenamento de conteúdo, da organização e gerenciamento da rede. Outras características também favoreceram o grande desenvolvimento dos sistemas

P2P. Estas características incluem computação distribuída na borda da rede, capacidade para execução de sistemas em larga escala, adaptação a populações dinâmicas de nodos (*peers*) e facilidade na criação de comunidades virtuais. Mesmo existindo diversas definições possíveis para os sistemas P2P, uma definição abrangente é feita por Theotokis & Spinellis em [57]:

"Sistemas Peer-to-Peer são sistemas distribuídos que consistem de nodos interconectados, com capacidade de se auto-organizar em topologias de rede, com o objetivo de compartilhar recursos como ciclos de CPU, armazenamento e banda, capazes de se adaptar a falhas e acomodar populações transientes de nodos, enquanto mantêm conectividade e desempenho aceitáveis, sem depender da intermediação ou suporte de uma autoridade central (servidor)."

A seguir são melhor apresentados os conceitos deste modelo de sistema e suas propriedades, as topologias e suas consequências, assim como algumas aplicações possíveis.

A principal característica de um *peer* é a capacidade de servir requisições de outros *peers*, ao mesmo tempo em que pode requisitar serviços de outros *peers* [53]. Desta forma, um sistema P2P não depende de uma máquina ou processo central, mas do conjunto de *peers* que o formam. A figura 2.2 ilustra a idéia da descentralização do processamento ou computação. O sistema é representado por uma nuvem de *peers* que interagem entre si e com os usuários.

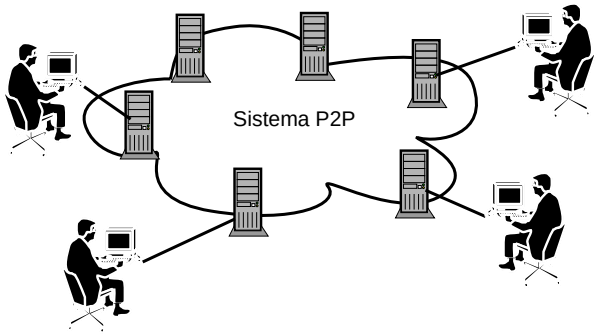


Figura 2.2: Processamento descentralizado nos *peers* do sistema.

Outras características marcantes são a troca ou compartilhamento direto de recursos

computacionais entre os *peers* e a habilidade para tratar a instabilidade e a conectividade variável, permitindo que o sistema seja potencialmente tolerante a falhas [57].

A falta de uma entidade que centralize totalmente as tarefas aumenta potencialmente a disponibilidade do sistema, pois há uma redundância de recursos intrínseca. Esta característica também permite que estes sistemas se adaptem a alterações no seu estado, pois quando um *peer* deixa o sistema, o restante dos *peers* podem se adaptar para manter o sistema ativo. A disponibilidade pode ser alta, pois os recursos e tarefas estão distribuídos por todo o sistema. Por outro lado, quanto mais a população de *peers* aumenta e mais recursos tornam-se disponíveis, maior é a necessidade por um gerenciamento eficiente destes recursos, para que o desempenho do sistema seja mantido.

Como anteriormente citado, um *peer*, também chamado de nodo neste trabalho, é todo componente conectado a uma rede, que possa compartilhar os seus recursos com outros *peers*, e também utilizar recursos de outros *peers*. Os *peers* conectados à uma rede como a Internet possuem configuração de hardware e software diferentes. Por esta razão, os recursos são o que cada nodo possui, como por exemplo processador, memória, espaço em disco, arquivos armazenados, entre outros.

Normalmente, todos os *peers* do sistema possuem as mesmas funcionalidades e podem realizar tarefas idênticas. Além disso, *peers* com maior capacidade computacional podem ser utilizados para tarefas específicas, como manter índices com informações sobre recursos disponíveis na rede, manter informações sobre caminhos entre os *peers*, prover serviços de autenticação de usuários, entre outros [57, 26]. Neste caso, estes *peers* especiais, chamados de *superpeers*, comportam-se como servidores para tais serviços e devem ser mantidos ou substituídos dinamicamente, para que a rede mantenha sua disponibilidade.

Entretanto, há sistemas ditos P2P e organizados de forma centralizada, com *superpeers* fixos. O Napster [45], por exemplo, utiliza nodos fixos para centralizar informações sobre os conteúdos publicados. Neste caso, apesar dos servidores serem pontos de vulnerabilidade da rede, restringindo a disponibilidade e escalabilidade, após obter as informações necessárias, os demais *peers* realizam a troca de informações diretamente entre si. Este também pode ser o caso de aplicações colaborativas, como as de troca de mensagens ins-

tantâneas, onde as listas de contatos ficam armazenadas em alguns nodos. Para participar do sistema, os demais nodos se autenticam, e recebem as informações necessárias sobre a sua lista de contatos. A partir desse ponto os nodos se comunicam diretamente.

Nas seções seguintes serão apresentadas diferentes organizações e aplicações dos sistemas *Peer-To-Peer*.

### 2.1.1 Organização dos Sistemas *Peer-To-Peer*

Os sistemas *Peer-To-Peer* possuem organizações distintas, de acordo com a topologia utilizada ou com o seu modo de operação. Os nodos normalmente estão dispersos em grandes áreas geográficas, possuem recursos heterogêneos e são administrados de maneira diferente. Para interligar estes recursos computacionais distribuídos, é necessária uma infra-estrutura que utilize protocolos independentes de plataforma e da estrutura física da rede. Ou seja, a topologia de rede utilizada por um sistema P2P deve ser independente da topologia física da rede.

Para isso, uma rede lógica (*overlay network*) [18] é criada utilizando a estrutura de protocolos presentes na rede, acima do protocolo da Internet (*Internet Protocol* - IP), ou mesmo sobre os protocolos da camada de transporte (TCP - *Transfer Control Protocol* ou UDP - *User Datagram Protocol*). O sistema P2P se baseia nesta rede para fazer a interação entre os processos (*peers*).

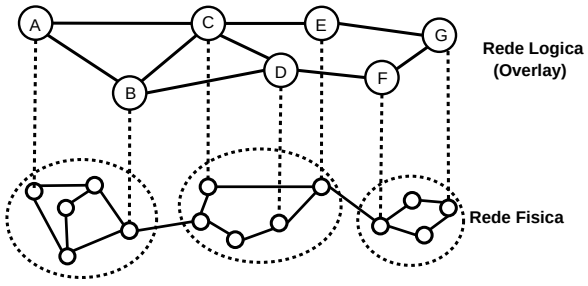


Figura 2.3: Exemplo de uma topologia de rede lógica acima da rede física.

A figura 2.3 mostra uma rede lógica, construída acima da rede física. A topologia criada depende da interação entre os nodos do sistema. Os caminhos de comunicação

(*links*) criados, quando transformados em caminhos reais, se basearão em endereços e rotas da rede física. Por exemplo, na rede lógica há um caminho direto entre os nodos B e D, por outro lado, na topologia física desta rede, não apenas não existe um caminho direto entre estes dois nodos, como eles não pertencem à mesma subrede.

Um sistema P2P pode ser organizado de maneira estruturada ou não estruturada [57, 51]. O sistema P2P não estruturado é aquele em que não há controle de como os *peers* ou recursos estão agrupados na rede, em outras palavras, os *peers* e recursos são inseridos na rede de maneira aleatória. Ao contrário, sistemas P2P estruturados controlam o modo e o local onde os *peers* e recursos são inseridos ou excluídos da rede, organizando-a de acordo com suas necessidades.

Nos sistemas estruturados normalmente são mantidas listas com dados ou mapas sobre recursos disponíveis e sua localização. Desta forma, buscas por recursos tendem a ser mais específicas e eficientes, através da consulta a estas listas. Nos sistemas não estruturados, os recursos normalmente têm que ser localizados sem auxílio de listas centralizadas. Para estas buscas, utiliza-se desde estratégias simples como a inundação [4], até buscas por caminhos aleatórios [42].

Em geral, sistemas não estruturados são mais indicados para populações onde os nodos se comportam de maneira muito dinâmica [57], com alta taxa de entrada e saída de nodos no sistema. Os sistemas estruturados, por sua vez, são mais apropriados para populações com menores taxas de variação, pelo custo de manutenção e organização das suas estruturas.

A seguir são apresentadas as organizações quanto à topologia adotada pelos sistemas P2P, baseadas em graus de centralização. Apesar da idéia principal dos sistemas P2P ser a decentralização das atividades, na prática diversos níveis de centralização são encontrados nos sistemas. Há desde sistemas "puros" ou totalmente descentralizados, até sistemas que se utilizam de nodos que centralizam grande parte das tarefas [8]. Este é o caso, por exemplo, do JXTA, que será descrito adiante.

#### 2.1.1.1 Organização Descentralizada

Os sistemas P2P com organização descentralizada, também chamados de "puros", são os mais simples, pois todos os nodos possuem a mesma funcionalidade [57, 53], ou seja, todos os nodos podem realizar as mesmas tarefas. Além disso, não há uma coordenação centralizada das tarefas, cada nodo é responsável por sua própria administração e manutenção.

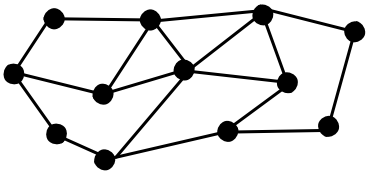

Figura 2.4: Exemplo de topologia descentralizada.

A figura 2.4 mostra um exemplo de topologia ou organização descentralizada, ou pura, para um sistema P2P. Os pontos representam os nodos ativos no sistema e as linhas entre eles representam os canais de comunicação.

A falta de coordenação centralizada apresenta vantagens e desvantagens. Por um lado, há a liberdade de cada nodo ser mantido e configurado como o administrador local desejar, sem a necessidade de um acordo entre os participantes da rede. Por outro, há a dificuldade em construir serviços que exijam coordenação, como processamento distribuído.

Uma rede com estrutura descentralizada possui baixa vulnerabilidade, pois o sistema em geral tem vários nodos que fornecem os mesmos serviços. Com isso, é necessário que grande parte dos nodos estejam indisponíveis para que o sistema não funcione adequadamente. Novamente, o desempenho do sistema pode ser ruim se não houver estratégias eficientes aplicadas à busca de recursos, pois as informações sobre os mesmos também estão espalhados por todo o sistema [59, 12].

O sistema para compartilhamento de arquivos Gnutella [25, 41] baseia-se em uma organização descentralizada e não estruturada. Além de todos os nodos possuírem a mesma funcionalidade, não há estruturas centralizadas com informações sobre localização destes e dos recursos presentes no sistema. A busca por recursos é feita por inundação.

Cada nodo envia as requisições de busca a todos os nodos aos quais está diretamente conectado no sistema (vizinhos). Os nodos que recebem uma requisição reenviam esta aos seus outros vizinhos, até um limite de pulos (*hops*), ou até que a busca seja completada com sucesso. Um sistema organizado desta forma permite o anonimato dos participantes e do conteúdo com maior facilidade, se comparado com sistemas que apresentam algum grau de centralização ou índices para buscas.

#### 2.1.1.2 Organização Parcialmente Centralizada

Os sistemas que adotam esta topologia têm base na estrutura descentralizada, como descrito na seção anterior, acrescida de nodos especiais, que possuem funcionalidades adicionais, e que são disponibilizadas a todos os *peers* do sistema. Esses nodos especiais, também chamados de "supernodos", "*superpeers*" ou "*rendezvous*" (ponto de encontro), agem, portanto, como servidores locais. As funcionalidades, ou serviços oferecidos pelos supernodos podem incluir: manter e gerenciar índices de *peers* e de conteúdo, prover serviços para autenticação de *peers*, manter informação sobre rotas entre nodos, entre outros [57, 26].

Para manter as características de escalabilidade e baixa vulnerabilidade, estes nodos são determinados e mantidos dinamicamente, de acordo com a população de *peers*. No caso de falha, ou de algum supernodo deixar de oferecer os serviços, este deve ser substituído de forma automática e dinâmica. Com esta estrutura, os *peers* do sistema acessam os serviços oferecidos pelo supernodo selecionado, formando subredes centralizadas pelos supernodos.

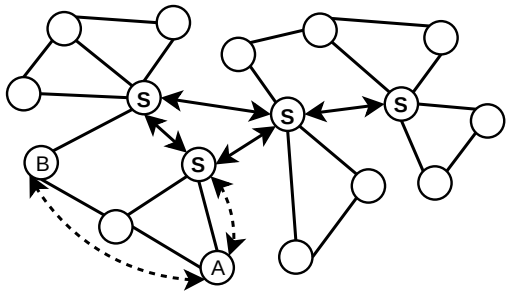


Figura 2.5: Exemplo de topologia híbrida.

A figura 2.5 mostra um exemplo de topologia ou organização híbrida de um sistema P2P. Os nodos rotulados com "S" são o que chamamos previamente de supernodos. Neste exemplo, o nodo A faz a requisição de um recurso ao supernodo ao qual está conectado. Este, por sua vez, localiza esta informação, e retorna ao nodo A, a localização deste recurso, no caso, representado pelo nodo B. A partir deste momento, o nodo A entra em contato diretamente com o nodo B.

Os sistemas para compartilhamento de arquivos eDonkey [20] e Kazaa [38] são organizados como sistemas parcialmente centralizados. Nestes dois sistemas, os supernodos são eleitos com base em sua capacidade computacional e largura de banda. O Kazaa utiliza os supernodos para manter listas de arquivos com a localização de cada um destes arquivos.

#### 2.1.1.3 Organização Centralizada

Nesta estrutura, há um supernodo central, ou um conjunto de supernodos, que mantém índices e outras informações para facilitar a interação dos *peers*, assim como na topologia híbrida. A grande diferença é a maneira como estes supernodos são mantidos. Na topologia centralizada, estes supernodos são estáticos, ou seja, não são substituídos dinamicamente em caso de falha e não mudam conforme a população de *peers*. Estes supernodos são pontos que deixam o sistema mais suscetível a falhas em geral, além de limitar a escalabilidade [57]. Ainda assim, alguns sistemas com organização centralizada são considerados P2P pela característica da comunicação direta entre os *peers*, após os supernodos terem fornecido as informações necessárias.

Com esta organização, o controle e a monitoração sobre recursos e conteúdos disponíveis é potencialmente mais fácil de ser realizado [25]. É também por este motivo que muitas aplicações, principalmente as de troca de conteúdo, são baseadas em estruturas descentralizadas, nas quais é mais fácil manter o anonimato dos usuários e o sigilo das informações compartilhadas entre os nodos.

No início, o sistema para compartilhamento de arquivos Napster [45, 41] mantinha listas em seus servidores, com informações sobre os arquivos (principalmente músicas) e a localização destes. A partir destas informações, cada nodo podia se conectar a outros para fazer o compartilhamento dos arquivos escolhidos. Devido a ações judiciais sobre direitos autorais dos arquivos disponibilizados, facilmente fiscalizados a partir dos servidores centrais, hoje este sistema exige pagamento para acesso ilimitado aos arquivos.

### 2.1.2 Aplicação dos Sistemas *Peer-To-Peer*

Os sistemas P2P têm sido intensamente desenvolvidos para a aplicação em diversas áreas como comunicação e colaboração, sistemas de bancos de dados distribuídos, computação distribuída e compartilhamento e distribuição de conteúdo.

A categoria de sistemas para comunicação e colaboração [15] baseados em P2P incluem aqueles que permitem o trabalho de forma colaborativa entre pessoas ou *peers*, mesmo distantes. Aplicações comuns incluem *chats* e troca de mensagens instantâneas, sistemas para edição de documentos em tempo real, jogos em rede, entre outros.

Os sistemas de troca de mensagens entre *peers* normalmente são construídos com a presença de supernodos, que mantêm a lista de contatos de cada *peer*. Geralmente, quando um *peer* inicia a sua execução, a lista com os seus contatos e respectivos endereços é fornecida pelo supernodo. Quando um *peer* requisita a comunicação com um outro, isso é feito diretamente, sem necessidade de que as mensagens trafeguem pelo supernodo que forneceu a lista de contatos.

Os sistemas de bancos de dados distribuídos também têm sido implementados baseando-se na infraestrutura de sistemas P2P. Um exemplo de aplicação é o Edutella [46], um projeto em código aberto, para prover infraestrutura para construção de bancos de dados

distribuídos para aplicações P2P.

A computação distribuída é outra aplicação para a tecnologia P2P. O objetivo deste tipo de aplicação é aproveitar o poder de processamento subutilizado dos nodos da Internet [44]. Para isso, as tarefas são particionadas e distribuídas entre os integrantes (nodos) do sistema, através da rede. Após o processamento individual de cada parte da tarefa por um nodo, estas tarefas são devolvidas, juntamente com os resultados obtidos. Para que sejam realizadas corretamente, geralmente é necessário que as etapas de particionamento, distribuição e obtenção dos resultados sejam coordenadas de forma centralizada. Essa característica normalmente exige a presença de supernodos, para fazer a coordenação das atividades.

Há grande semelhança entre a computação distribuída baseada em P2P com a computação em *grid* [23] , pois ambas têm foco no compartilhamento de recursos em larga-escala. A tendência é de que as tecnologias P2P sejam utilizadas para a construção de aplicações para computação em *grid*, com a convergência destas duas áreas [24].

Um exemplo de sistema P2P para computação distribuída é o Seti@Home [1], que faz o processamento distribuído de sinais captados do espaço, para a busca de vida extraterrestre. Este sistema possui um nodo central que coordena as atividades dos *peers*. Após processar os sinais, cada *peer* retorna os resultados ao nodo coordenador. Neste sistema, há comunicação somente entre o nodo central e cada um dos *peers* que executam as tarefas.

A aplicação de sistemas P2P para o compartilhamento de conteúdo é a mais popular. Foi devido a esta aplicação que sistemas P2P começaram a ser desenvolvidos com maior intensidade. Todos os sistemas que oferecem serviços ou infraestrutura para a troca de conteúdo entre usuários estão nesta categoria. Os sistemas se referem a aplicativos que permitem a distribuição de conteúdo, enquanto a infraestrutura está relacionada a *frameworks* ou bibliotecas, que provêem a base para a construção de aplicações P2P.

Os sistemas de distribuição de conteúdo podem ser extremamente simples, com organização descentralizada e busca por conteúdo baseadas nas técnicas de busca em largura ou inundação (*flooding*), como é o caso do Gnutella [25]. Em contrapartida, há sistemas

sofisticados, que utilizam armazenamento distribuído de índices e conteúdos, utilizando-se também de supernodos. Estes sistemas geralmente implementam algoritmos avançados para obter maior eficiência na busca, obtenção, publicação e edição de conteúdos espalhados pela rede.

Além dos já citados ao longo do texto, como exemplos de sistemas para distribuição e compartilhamento de conteúdo há o Napster [45], Kazaa [38], Gnutella [25], eDonkey [20], eMule [21], e mais recentemente, o BitTorrent [7]. O BitTorrent é um sistema cujo objetivo é aumentar a rapidez e eficiência do compartilhamento de arquivos. Basicamente, este sistema particiona um arquivo em vários arquivos menores e os distribui entre os participantes do sistema ou rede virtual. Quando um *peer* requisita este arquivo, os fragmentos são enviados por *peers* diferentes, de maneira paralela.

#### 2.1.2.1 Propriedades dos Sistemas *Peer-To-Peer*

Os sistemas P2P têm sido muito utilizados principalmente devido à sua capacidade de permitir o compartilhamento do poder de armazenamento e processamento espalhado pelas redes. Mas, além disso, há diversas propriedades que são desejáveis em um sistema P2P. Estas propriedades incluem desempenho, segurança, escalabilidade, disponibilidade, capacidade de gerenciamento e justiça no compartilhamento dos recursos.

As propriedades dos sistemas P2P, assim como a maneira como os *peers* interagem - publicam, pesquisam e trocam arquivos - são influenciados pela organização e topologia do sistema [57, 44]. Por exemplo, um sistema parcialmente centralizado poderá utilizar algoritmos avançados para organizar suas estruturas, tornando mais rápida a publicação e pesquisa de conteúdos e recursos. Este sistema também poderá oferecer serviços de autenticação de usuários, ajudando a garantir a autenticidade, privacidade e confidencialidade de conteúdos, itens que fazem parte da questão da segurança da comunicação entre *peers*.

A segurança de um sistema inclui: integridade e autenticidade, que é a capacidade de assegurar que o conteúdo é completo, correto, e de fonte segura; privacidade e confidencialidade, que significa que os dados estarão acessíveis somente a usuários autorizados; além da disponibilidade, ou a capacidade de que um nodo tenha acesso a arquivos, informações

e recursos, quando requisitado.

A capacidade de gerenciamento dos recursos pode ter grandes variações. Um sistema pode prover apenas serviços básicos, como publicação, busca e recuperação de arquivos, ou então, prover além destes, serviços avançados como editar e remover arquivos, armazenados de maneira distribuída. O desempenho, medido normalmente pelo tempo necessário para que um nodo realize os serviços disponibilizados, deve ser mantido independentemente da variação da população de nodos. A disponibilidade, que é um atributo de um sistema seguro, depende da escalabilidade do sistema.

O senso de justiça em um sistema P2P pode ser observado pela medida entre quantidade de recursos que um nodo disponibiliza e o que este mesmo nodo consome [3]. Para que os usuários participem da comunidade, ou sejam colaborativos, este senso de justiça deve ser conseguido. Nodos com maior capacidade de banda naturalmente têm mais requisições pelo compartilhamento dos seus arquivos e recursos, portanto também devem conseguir mais facilmente que um conteúdo ou recurso lhe seja disponibilizado.

## 2.2 A Plataforma JXTA

O JXTA [35] começou como um projeto mantido pela empresa *Sun Microsystems, Inc.* [55]. Hoje este projeto é desenvolvido por uma comunidade heterogênea de colaboradores. O nome deste projeto é derivado da palavra *juxtapose* (justapor), no sentido que o paradigma P2P está lado a lado com o modelo cliente-servidor [36].

A tecnologia JXTA é um conjunto de protocolos abertos, baseados no conceito P2P, cujo objetivo é fornecer interoperabilidade entre os diversos dispositivos de uma rede, independentemente da plataforma em que este dispositivo foi desenvolvido. Em outras palavras, permitir a comunicação entre quaisquer dispositivos conectados à Internet [35]. Para que o objetivo seja atingido, o JXTA provê as funcionalidades necessárias para a implementação de um sistema P2P, como: comunicação entre nodos; serviço de publicação e busca por nodos e demais recursos; organização de grupos; serviços para monitoração da rede.

O projeto JXTA foi desenvolvido inicialmente em Java e atualmente possui uma versão

em C. Independentemente disso, o desenvolvedor pode implementar seus próprios componentes ou protocolos em outras linguagens. Os sistemas baseados no JXTA podem também ser construídos sobre diferentes protocolos de transporte como o TCP, UDP e HTTP (*HyperText Transfer Protocol*) [60].

Cada processo que executa os protocolos do JXTA é chamado de nodo JXTA ou JXTA *peer*, e possui um identificador único utilizado para sua interação com outros nodos, no sistema JXTA. Um mesmo dispositivo ou máquina pode participar da comunidade virtual com vários nodos ativos. A partir do momento em que um nodo inicia a execução de um sistema baseado no JXTA, este nodo pode utilizar todos os serviços disponibilizados pela plataforma. Para a comunicação e colaboração entre os nodos JXTA, esta plataforma provê a formação de uma rede virtual, que permite a interação mesmo quando alguns nodos estão protegidos por *firewalls* ou NAT (*Name Address Translation*) [54].

As próximas seções descrevem a plataforma JXTA, mostrando os componentes e sua interação.

### 2.2.1 Arquitetura da Plataforma JXTA

Como mostrado na figura 2.6, a arquitetura do JXTA é organizada em três camadas: o núcleo, a camada de serviços e a camada de aplicações, descritas a seguir.

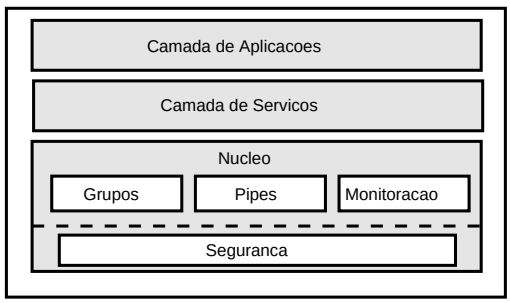


Figura 2.6: Arquitetura em camadas do JXTA.

O núcleo da plataforma JXTA encapsula e implementa os protocolos essenciais a qualquer aplicação P2P. Estes protocolos incluem serviços de criação de nodos, grupos e

canais de comunicação, descoberta ou busca por recursos e primitivas básicas de segurança, como será visto adiante.

A camada de serviços é formada por funções não essenciais disponibilizadas pelo JXTA. Esses serviços incluem sistemas de indexação para facilitar buscas por conteúdos, sistemas de armazenamento, compartilhamento de arquivos, sistemas de arquivos distribuídos, autenticação e serviço de Infraestrutura de Chaves Públicas (PKI-*Public Key Infraestructure*). Apesar de desejáveis, a implementação e utilização destes serviços é opcional.

Finalmente, a camada de aplicações inclui a implementação de serviços integrados, como compartilhamento de arquivos e recursos, troca de mensagens instantâneas, sistemas de leilão eletrônico, entre outros.

Os componentes que formam as camadas do JXTA interagem para formar as aplicações. Cada nodo JXTA implementa o núcleo da arquitetura JXTA e serviços que desejar utilizar das outras camadas, conforme a necessidade. Os componentes e conceitos definidos pela plataforma serão vistos a seguir.

### 2.2.2 Conceitos e Componentes da Plataforma JXTA

A rede JXTA é formada por *nodos JXTA*, que podem se organizar em *grupos* para prover serviços ou executar tarefas. Um nodo é qualquer processo, em um dispositivo conectado à rede, que implementa os protocolos da plataforma JXTA e que possui capacidade para se comunicar com outros nodos [26, 8]. Cada nodo é independente de todos os outros e possui um identificador único (ID) fornecido pela plataforma JXTA.

Além dos nodos, há outros componentes que contribuem para a construção dos sistemas P2P baseados no JXTA: *grupos de nodos*, *canais de comunicação* ou *pipes*, *mensagens*, *anúncios* ou *advertisements* e *interfaces* ou *endpoints*.

Quando inicia a execução de um sistema P2P JXTA, um nodo automaticamente torna-se parte do *grupo de nodos* principal do JXTA. Este nodo pode então criar novos grupos ou entrar em grupos já estabelecidos. Cada grupo é uma coleção de nodos que geralmente oferecem um conjunto comum de serviços e pode estabelecer suas próprias políticas de entrada. O primeiro critério para entrada de um nodo em um grupo normalmente é a

implementação, por este nodo, dos mesmos serviços que o restante dos nodos do grupo disponibilizam.

O grupo principal de nodos do JXTA é o grupo padrão, ao qual todos os nodos pertencem. Mas isto não significa que todos os nodos estão conectados através deste grupo, pois a rede formada pelos nodos JXTA possui topologia arbitrária. Uma decorrência deste fato é que não há necessariamente um caminho de comunicação entre todos os nodos, pois a rede pode estar particionada.

A plataforma JXTA fornece um conjunto padrão de serviços, não obrigatórios, que são normalmente implementados por todos os grupos e utilizados pelos nodos que o formam:

- Serviço de descoberta (*Discovery Service*), utilizado para a procura de recursos, como nodos e grupos, canais de comunicação e outros serviços;
- Serviço de gerenciamento do grupo (*Membership Service*), utilizado pelos membros para aceitar ou rejeitar requisições de entrada no grupo e manter o grupo de nodos;
- Serviço de acesso (*Access Service*), para verificar se uma requisição feita por um membro a outro membro do grupo pode ser atendida;
- Serviço de canais de comunicação (*Pipe Service*), utilizado para a criação e manutenção dos canais de comunicação entre os membros do grupo;
- Serviço de resolução (*Resolver Service*), para enviar requisições a outros membros do grupo, como pedir informações sobre um recurso.
- Serviço de monitoração (*Monitoring Service*), usado para permitir que os nodos monitorem outros membros do grupo.

Os nodos divulgam seus recursos e serviços através da publicação ou envio de *anúncios* para a rede JXTA. Os anúncios são documentos em formato XML (eXtensible Markup Language) [19], que contém as informações necessárias para que os nodos consigam utilizar o recurso publicado. Todos os componentes ou recursos da rede JXTA são representados por anúncios: nodos, grupos, *pipes* e serviços possuem anúncios que os descrevem, e que devem ser publicados para que os nodos os conheçam [61].

A comunicação entre os nodos é feita através de *pipes*, ou canais de comunicação criados pelos nodos para a troca de *mensagens*. Os *pipes* estão associados a *interfaces*, que por sua vez, estão associadas ao endereço físico do nodo (conjunto endereço IP e porta TCP). Entretanto, interfaces podem ter este valor alterado, caso o endereço físico do nodo sofra alterações. Esta mudança não depende de uma estrutura central ou serviço de nomes, como é o caso do DNS (Domain Name System) [47]. A alteração da associação do endereço físico á interface JXTA é feita de forma dinâmica, por um protocolo JXTA que será descrito nas próximas seções.

Os canais de comunicação normalmente são assíncronos (mensagens podem ser enviadas e recebidas a qualquer momento), unidirecionais e não confiáveis. Canais de comunicação também podem ser ponto a ponto (*point-to-point pipes*) ou de propagação (*propagate pipes*). Além destes, a plataforma JXTA oferece um canal de comunicação unidirecional que implementa algumas primitivas de segurança (*unicast secure pipe*), um canal de comunicação bidirecional (*bidipipe*), e o JXTA *Sockets* [36]. O *unicast secure pipe*, o *bidipipe* e o JXTA *Sockets* foram construídos a partir dos *pipes* ponto a ponto ou de propagação. A figura 2.7 ilustra os canais de comunicação ponto a ponto e de propagação.

Os canais de comunicação ponto-a-ponto são os que conectam dois pontos ou nodos. Com estes canais um nodo recebe as mensagens em seu ponto de entrada (*input pipe*), e o outro nodo as envia com seu ponto de saída (*output pipe*). Ainda, apesar do nome ponto-a-ponto, é possível que vários nodos que possuem um ponto de saída (*output pipe*), se conectem e enviem mensagens a um nodo através de um único ponto de entrada (*input pipe*) deste nodo.

Os canais de comunicação para propagação são aqueles em que um único ponto de saída de um nodo está conectado a pontos de entrada de outros nodos. Desta maneira, todos os nodos que possuírem estes pontos de entrada receberão as mensagens enviadas por um nodo, através de seu único ponto de saída.

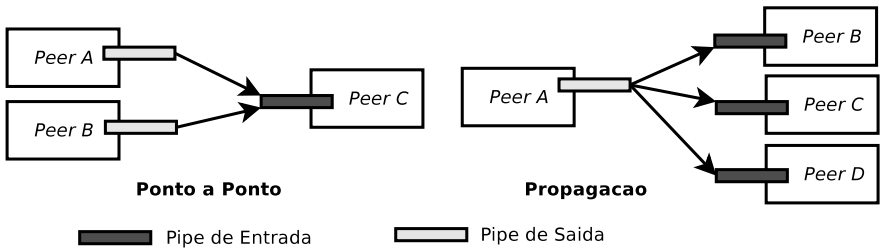


Figura 2.7: Canais de comunicação ponto-a-ponto e propagação.

Voltando às *mensagens* trocadas, estas são compostas de objetos enviados e recebidos pelos nodos. As mensagens contêm cabeçalhos dos protocolos utilizados na comunicação. Os cabeçalhos encapsulam os objetos enviados. As mensagens também possuem informações sobre origem, destino e rota das mesmas. Os objetos podem ser representados em código binário ou por documentos XML e podem conter qualquer tipo de informação. Os documentos contêm todas as informações necessárias para a correta interpretação da mensagem pelo nodo receptor.

### 2.2.3 Comunicação entre Nodos

A rede virtual do JXTA pode ser formada por nodos de diversos tipos, que podem implementar serviços variados, conforme sua capacidade. Por este motivo, e para melhor organizar a estrutura da rede virtual, os nodos são separados em quatro categorias [26], *minimal edge peer*, *full-featured edge peer*, *rendezvous peer* e *relay peer*, descritas a seguir.

A categoria de nodos (*minimal edge peer*) possui o mínimo possível de protocolos implementados, apenas para ser capaz de enviar e receber mensagens. Estes nodos normalmente estão em dispositivos com recursos limitados, como por exemplo telefones celulares.

A categoria (*full-featured edge peer*), além de enviar e receber mensagens, possui a capacidade de armazenar os anúncios recebidos. Esses nodos podem responder a requisições de pesquisas, com base nos anúncios armazenados, mas não propagam requisições aos seus nodos vizinhos.

Os nodos responsáveis por propagar as requisições de pesquisa entre os membros de

um grupo são chamados de nodos *rendezvous* (*rendezvous peer*). Além da lista de nodos do grupo ao qual pertence, este nodo mantém também uma lista de outros nodos *rendezvous*. Com esta lista, os nodos *rendezvous* propagam as requisições a outros *rendezvous*, que por sua vez propagam-nas aos nodos do seu grupo e a outros *rendezvous*. Este procedimento ocorre até que a requisição seja atendida, ou que o limite de busca da requisição seja atingido. O limite é a quantidade de pulos (*hops*) ou a quantidade de propagações iniciadas. Se a informação ou recurso requisitado for encontrado, o nodo que o possui responde diretamente ao nodo que fez a requisição, geralmente utilizando o caminho contrário feito pela requisição. O JXTA é capaz de detectar ciclos para que a requisição não trafegue indefinidamente pela rede. Este mecanismo tende a facilitar a busca por recursos dentro da rede JXTA.

Todo grupo de nodos deve possuir pelo menos um nodo *rendezvous*, e todo outro nodo do grupo deve estar conectado ao menos a um *rendezvous*. Ao entrar em um grupo, um nodo automaticamente procura e tenta se conectar a um *rendezvous*. Se não conseguir, o nodo automaticamente se torna um *rendezvous*.

O quarto tipo inclui os nodos que fornecem os serviços para roteamento de mensagens entre nodos e formação de caminhos: são chamados de nodos *relay* (*relay peer*). Estes nodos são responsáveis por manter informações sobre caminhos ou rotas, além de encaminhar as mensagens para os nodos corretos. Além disso, é esta categoria que faz a ligação entre nodos que estão atrás de *firewalls* ou NATs com o restante da rede virtual JXTA. Entretanto, estes nodos necessitam de uma conexão com um nodo *relay* externo ao *firewall* ou NAT, para manter a comunicação com toda a rede JXTA.

Nodos atrás de um *firewall* geralmente conseguem enviar mensagens para nodos fora do *firewall*, enquanto os nodos de fora normalmente não conseguem iniciar a comunicação com os nodos protegidos por *firewalls* [8]. Para que a comunicação entre estes nodos seja possível, é necessário que pelo menos um nodo do grupo de nodos atrás do *firewall* conheça um nodo *relay* de fora do *firewall*. Mais ainda, estes nodos precisam utilizar um protocolo que consiga atravessar o *firewall*, geralmente o HTTP.

Um exemplo da comunicação entre nodos separados por *firewalls* é mostrado na figura

2.8, onde os nodos A e B desejam se comunicar. Neste caso, ambos estão protegidos por *firewalls*. Para que esta comunicação seja estabelecida, é necessário que o nodo A inicie uma comunicação com um nodo *relay* externo ao *firewall* (nodo *relay* 1), utilizando um protocolo que atravesse o *firewall*. O nodo B também precisa ter iniciado sua comunicação com um nodo *relay* externo (*relay* 2). Os nodos *relay* 1 e 2 estabelecem um canal de comunicação entre si, utilizando qualquer protocolo de transporte, através da rede virtual do JXTA. A partir deste momento, os nodos A e B podem se comunicar, através da conexão virtual estabelecida.

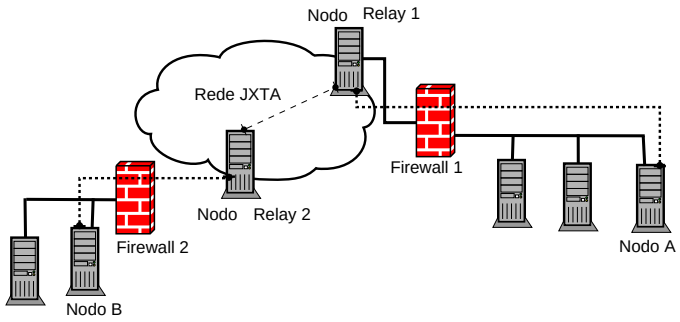


Figura 2.8: Comunicação através de *firewalls*.

Muitos nodos estão constantemente entrando e saindo da rede virtual JXTA, e tantos outros possuem endereços IP dinâmicos. Por esta razão, não é possível utilizar o endereço IP do dispositivo como seu identificador na rede JXTA. Para solucionar este problema, foram criados identificadores únicos para cada componente ou recurso da rede JXTA. Nodos, canais, anúncios, grupos e outros componentes possuem, cada um, um identificador para que possam ser acessados de maneira única. Por exemplo, para que um nodo acesse um recurso, basta que o nodo conheça o identificador deste recurso.

A próxima questão é a maneira como os nodos resolvem ou associam o identificador virtual ao endereço do componente, dado que não é possível centralizar estas informações. A centralização não é possível pela característica dinâmica da rede virtual, onde dispositivos podem conectar e desconectar frequentemente. Para solucionar este problema, o JXTA implementa um protocolo de associação que permite que um identificador do

JXTA seja associado ao endereço físico do nodo, em tempo de requisição, ou seja, após uma requisição ser enviada e conforme esta requisição caminha na rede. Sempre que uma mensagem deve ser enviada a um nodo, este protocolo associa o identificador único do nodo na rede JXTA ao endereço deste nodo na rede física.

A próxima seção descreve este e os outros protocolos fornecidos pela plataforma JXTA, que são a base do funcionamento da rede virtual.

### 2.2.4 Protocolos da Plataforma JXTA

Os protocolos que formam a base da plataforma JXTA são o Protocolo de Descoberta (PDP-*Peer Discovery Protocol*), o Protocolo de Informações (PIP-*Peer Information Protocol*), Protocolo de Resolução (PRP-*Peer Resolver Protocol*), Protocolo de Associação de Canais de Comunicação (PBP-*Pipe Binding Protocol*), Protocolo de Roteamento (ERP-*Endpoint Routing Protocol*) e o Protocolo de *Rendezvous* (RVP-*Rendezvous Protocol*) [26, 36], descritos brevemente a seguir.

O PDP ou Protocolo de Descoberta é utilizado pelos nodos para publicar anúncios sobre os seus recursos e procurar recursos disponibilizados por outros nodos. Estes recursos podem ser grupos, canais, serviços ou qualquer outro recurso que tenha um anúncio publicado. Este protocolo é implementado pelos grupos, o que significa que um nodo precisa estar em um grupo para utilizá-lo. As requisições de pesquisa ou descoberta de recursos são feitas primeiramente no contexto do grupo e depois são repassadas à rede JXTA com a utilização de nodos *rendezvous*. Apesar deste protocolo ser o padrão, os grupos podem melhorá-lo e até implementar outras formas para a busca por recursos.

O PIP, Protocolo de Informações, é utilizado para obter informações sobre o estado e a capacidade dos nodos. Este protocolo fornece um conjunto de mensagens que podem ser enviadas para obter informações específicas sobre outros nodos conhecidos. Por exemplo, podem ser enviadas mensagens para verificar se determinado nodo está ativo. Outra aplicação é o controle do tráfego de mensagens pelos nodos. Com informações sobre a carga de trabalho dos nodos, o roteamento de mensagens pode ser feito utilizando caminhos mais eficientes.

O PRP ou Protocolo de Resolução é o serviço utilizado para resolver requisições genéricas. Em outras palavras, este protocolo é responsável pelo envio de requisições ou consultas aos nodos da rede e para a obtenção das respostas. Este protocolo permite que os nodos definam e troquem qualquer tipo de informação. As requisições podem ser feitas a nodos específicos ou a grupos inteiros. Entretanto, este protocolo não garante que sejam obtidas respostas, nem que estas retornem corretamente ao nodo emissor da requisição [8]. O PRP também é a base para a implementação dos protocolos de Informações (PIP) e Descoberta (PDP).

O protocolo utilizado para fazer a associação entre a interface do canal de comunicação com o seu respectivo anúncio, é o PBP ou *Pipe Binding Protocol*. Os canais de comunicação virtuais estabelecidos entre os nodos podem ser formados ou compreender diversos outros nodos. Como nodos podem conectar e desconectar constantemente, as duas pontas ou interfaces dos canais de comunicação precisam manter adequadamente a conexão, restabelecendo-a caso algum nodo do canal se desconecte. Além desta tarefa, o PBP também é responsável pela criação e manutenção das conexões entre os nodos separados por *firewalls* ou NATs. Para realizar suas tarefas, este protocolo utiliza o Protocolo de Roteamento descrito a seguir.

Para que os canais de comunicação sejam estabelecidos e as mensagens possam ser trocadas entre os nodos, é necessário traçar o seu caminho ou rota na rede virtual. Esta tarefa é realizada pelo ERP ou Protocolo de Roteamento, implementado nos nodos *relay*. Com isso, quando um nodo necessita enviar uma mensagem a outro, o primeiro nodo verifica se possui as informações de roteamento armazenadas localmente. Caso as informações de roteamento não sejam encontradas, este nodo envia uma requisição ao nodo *relay*, que se encarrega de encontrar um caminho até o nodo destino. A informação sobre a rota é uma sequência de nodos por onde a mensagem deve passar. Esta informação acompanha a mensagem, para que os nodos saibam para onde devem encaminhar aquela mensagem, até ser entregue ao destino correto.

Por último, o Protocolo de *Rendezvous* é responsável por propagar as mensagens dentro de um grupo de nodos. A propagação de mensagens pode ser controlada pelos nodos,

através da limitação do número de propagações que podem ser iniciadas e da detecção de ciclos (*loopbacks*). O Protocolo de *Rendezvous* utiliza o Protocolo de Roteamento (ERP) para localizar nodos no grupo e determinar as rotas das mensagens. Os protocolos de Resolução (PRP) e de Associação de Canais (PBP) utilizam este protocolo para propagar as suas mensagens.

### 2.2.5 Mecanismo de Segurança JXTA

Entre os requisitos de segurança desejáveis em um sistema P2P, a plataforma JXTA permite alcançar: confidencialidade, garantindo que o conteúdo de mensagens não sejam vistos por nodos sem autorização; autenticidade, garantindo a origem da mensagem e a autorização para o envio; e integridade, garantindo que mensagens não terão seu conteúdo alterado indevidamente até chegarem ao destino. Uma forma de implementar requisitos de segurança é fazer o controle diretamente sobre as mensagens ou os dados trafegados pela rede. Outra opção é garantir que o canal de comunicação por onde as mensagens trafegam seja seguro [37].

O JXTA oferece meios para a implementação de segurança para os dados trafegados através do emprego de criptografia de chave pública e criptografia de chave secreta, além de serviços de autenticação para entrada de nodos em grupos, possibilitando também a construção de grupos seguros. Adicionalmente, para garantir confidencialidade e integridade de canais de comunicação, é possível a utilização de protocolos de transporte seguros.

Para atingir os requisitos de segurança sobre as mensagens, sem a presença de uma conexão segura, geralmente é necessário que cada mensagem carregue informações adicionais. Essas informações podem ser credenciais do JXTA (*credentials*), resumos digitais da própria mensagem, certificados e chaves públicas.

Resumos digitais ajudam a garantir a integridade de uma mensagem, enquanto o uso de criptografia sobre o conteúdo, deve garantir a confidencialidade. As credenciais também são inseridas dentro de cada mensagem e são compostas de informações sobre o nodo origem da mensagem. Essas informações são utilizadas para verificar se os dados sobre o

nodo origem, presentes no cabeçalho da mensagem, estão corretos. Com isso, credenciais podem ser utilizadas para garantir a autenticidade da mensagem e a autorização de um nodo para enviar a mensagem considerada. Mais ainda, credenciais também são úteis para a construção de grupos seguros, através do controle de acesso aos próprios grupos e aos serviços oferecidos por estes.

A plataforma JXTA também oferece algumas opções para a implementação de segurança de canais de comunicação baseados em tecnologias já existentes, como é o caso do *Transport Layer Security*, que permite o estabelecimento de conexões seguras entre pares de nodos, inclusive com o uso de criptografia [61].

O JXTA permite a construção de grupos de *peers* seguros, que exigem autenticação para a entrada no grupo. A forma mais comum de autenticação é a utilização de login e senha, mas cada grupo do JXTA pode definir seus próprios métodos de autenticação. Os grupos seguros implementam um módulo que gera credenciais para o grupo. Para participar de um grupo seguro, os *peers* também devem implementar este módulo. As credenciais contêm as informações que permitem que um *peer* se comunique com o grupo (como o login e a senha). O *peer* que criou o grupo normalmente é o responsável por manter as credenciais e fazer a autenticação do *peer* para a entrada no grupo. O grupo pode permitir a criação de novos *logins* e senhas pelos *peers* durante a execução, ou ter *logins* e senhas pré-definidos. No último caso, o usuário deve saber previamente o login e senha para poder participar do grupo.

# Capítulo 3

# Sistemas Distribuídos Confiáveis

Os sistemas distribuídos têm se tornado importantes para organizações e indivíduos. Para evitar o custo significativo de possíveis falhas, é importante garantir a disponibilidade destes sistemas. Um sistema capaz de evitar que as falhas de alguns componentes afetem o funcionamento do sistema como um todo é dito um sistema tolerante a falhas. Esta tolerância pode ser conseguida com a utilização de estratégias que aproveitam a redundância inerente aos sistemas distribuídos. Neste capítulo são apresentadas estratégias clássicas para a implementação de sistemas distribuídos com alta disponibilidade. Inicialmente são apresentados os principais modelos de falhas utilizados para sistemas distribuídos, seguidos de propriedades de tolerância a falhas. A seguir são descritos os protocolos de acordo, incluindo uma definição de consenso e difusão atômica, e os protocolos de confirmação, com destaque para o clássico algoritmo de confirmação em duas fases (*Two-Phase Commit*). Por fim são apresentadas estratégias de replicação e o serviço de gestão de grupos.

## 3.1 Modelos de Falhas e Sistemas Distribuídos

Para a construção de sistemas distribuídos confiáveis, é necessário modelar o comportamento dos seus componentes quanto às falhas que podem ocorrer, e quanto à forma de interação destes componentes. Os sistemas distribuídos têm seu funcionamento baseado na interação de vários processos, que se comunicam através de trocas de mensagens utilizando canais de comunicação. Os processos e canais de comunicação, que são os componentes

do sistema, podem falhar e se recuperar de uma falha a qualquer momento. Quando um componente possui um comportamento fora da sua especificação, diz-se que este componente sofreu uma falha [31, 30], e de maneira análoga, um comportamento previsto na especificação caracteriza um componente com funcionamento correto, ou sem-falhas.

As falhas que podem ocorrer em processos são falhas por *parada*, por *omissão*, *falhas de desempenho* e *falhas bizantinas* [31, 28]. As falhas por parada, também conhecidas como *crash*, são um tipo de falha em que os processos deixam de executar todas as ações, tornando-se completamente inativos. As falhas por omissão ocorrem quando um processo não executa algumas das ações esperadas. As falhas de desempenho ocorrem quando um processo não executa uma ação em um certo tempo previsto. Essas falhas são ditas benignas, pois até o momento da falha, os processos executam dentro da especificação. Por último, caracteriza-se falha bizantina quando algum processo age de maneira não prevista, executando ações arbitrárias. Estas falhas também são chamadas de malignas [39, 31]. A classe de falhas bizantinas engloba todas os outros tipos de falhas.

Canais de comunicação podem ser classificados como *confiáveis*, *não confiáveis* e *com perdas equitáveis* de mensagens. Canais de comunicação confiáveis não permitem que mensagens se percam ou sejam corrompidas, seja por mecanismos de retransmissão ou codificação. Contudo, mesmo canais de comunicação confiáveis podem parar de funcionar, por problemas físicos por exemplo. Tais falhas podem causar o isolamento de processos em subgrupos, cada processo em uma partição diferente do sistema. Por outro lado, canais não confiáveis admitem perdas de mensagens de forma que não é possível fazer alguma hipótese sobre a ocorrência destas. Finalmente, os canais de comunicação com perdas equitáveis de mensagens (*fair lossy*), admitem perdas de mensagens de maneira equitativa. Ou seja, se um processo *pi* envia uma infinidade de vezes, uma mensagem *m* a um processo correto *pj*, *pj* recebe *m* uma infinidade de vezes.

Levando em consideração características temporais, um sistema pode ser classificado como *síncrono* ou *assíncrono* [43, 31]. No modelo síncrono é possível estabelecer limites de tempo para a execução das tarefas e para a transmissão de mensagens. Com este modelo, é possível detectar falhas de processos e perda de mensagens com precisão. No

modelo assíncrono ocorre exatamente o contrário. Não é possível estabelecer limites de tempo para a execução de tarefas e para a transmissão de mensagens [22]. Com isso, torna-se impossível diferenciar com precisão um processo que parou de executar (falho), de um processo extremamente lento, justamente pelo fato de não haver limites temporais no sistema. Diversos modelos parcialmente síncronos também têm sido definidos [43] com o objetivo de refletir de forma mais precisa o comportamento de sistemas reais.

### 3.1.1 Tolerância a Falhas: Propriedades

As estratégias de tolerância a falhas visam aumentar a disponibilidade dos sistemas distribuídos, ou seja, manter o sistema disponível com comportamento correto, apesar das falhas. Diversas propriedades podem ser consideradas para atingir os requisitos necessários aos sistemas distribuídos. Dentre estas propriedades, a segurança (*safety*), a progressão (*liveness*) e a pontualidade (*timeliness*) são as mais importantes [32, 28].

A propriedade de segurança em um sistema assegura, informalmente, que este sistema realiza as tarefas corretamente durante sua execução. Esta propriedade engloba outras, como por exemplo *reliability* e *availability*. *Reliability* é a capacidade de um sistema funcionar de acordo com sua especificação, ininterruptamente, durante um intervalo de tempo. *Availability* é a capacidade de um sistema em responder de acordo com sua especificação, durante um intervalo de tempo no qual possivelmente há falhas seguidas de recuperações.

A propriedade de progressão garante que, ao final, o sistema produzirá resultados corretos. Em outras palavras, a progressão garante que as tarefas evoluem ao longo da execução do sistema. Para que ao final da execução o sistema produza resultados, o sistema deve também apresentar a propriedade de *terminação*. A *terminação* garante que a execução de uma tarefa progride até seu estado final ou conclusão.

Estas propriedades caracterizam a disponibilidade do sistema, que de forma geral pode ser vista como o tempo em que um sistema está disponível para utilização. Não há como garantir sempre a disponibilidade total, dado que é possível que todos os componentes de um sistema falhem ao mesmo tempo, causando uma parada total no serviço oferecido.

Apesar disto, se mesmo na presença de falhas de alguns componentes, um sistema satisfaz sua especificação, este sistema é tolerante a falhas.

Finalmente, a pontualidade é uma propriedade temporal, que caracteriza o tempo no qual as tarefas são realizadas pelo sistema. Ou seja, através da pontualidade, é possível caracterizar o sistema, quanto ao tempo necessário para realizar tarefas.

## 3.2 Protocolos de Acordo ou Consenso

A necessidade de protocolos de acordo advém das características dos sistemas distribuídos, nos quais os processos precisam cooperar para realizar as tarefas, com o intuito de garantir a progressão do sistema e a corretude dos resultados. É necessário que os processos se comuniquem de forma a manter o sistema em um estado consistente. Os protocolos de acordo podem ser empregados para resolver problemas de cooperação, conhecidos como *problemas de acordo*, que envolvem um consenso sobre um valor a ser considerado ou ação a ser executada pelos processos, em determinado momento no sistema [58].

O consenso pode ser considerado a base para os problemas de acordo e pode ser descrito sucintamente como a seguir. Cada processo participante de um grupo ou sistema deve propor um valor e todos os processos corretos neste mesmo sistema devem *decidir* por apenas um valor dentre os propostos, mesmo que alguns processos do sistema estejam falhos. De maneira geral, o consenso é realizado em duas fases, a primeira para o envio dos valores e a segunda fase para a confirmação.

Os algoritmos de resolução do consenso devem garantir três propriedades: acordo, validade e terminação. O acordo se refere ao fato de que todos os processos, corretos ou não, decidem pelo mesmo valor. A validade garante que se um processo decide por um valor, então este valor foi proposto por algum processo. A terminação se refere ao fato de que todo processo decide por um valor, em uma quantidade finita de passos [11, 58]. Quando estas propriedades se aplicam tanto a processos falhos como corretos, o algoritmo é uniforme, quando as propriedades se aplicam apenas a processos corretos, o algoritmo é dito não uniforme.

A dificuldade da resolução do consenso depende do tipo de sistema e modelo de falhas

considerado. Em um sistema síncrono onde falhas por parada podem ser facilmente detectadas, o consenso pode ser alcançado. Para sistemas assíncronos, um resultado conhecido é o chamado resultado de impossibilidade de FLP [22], de Fischer, Lynch e Paterson, segundo o qual em um ambiente assíncrono em que é possível a ocorrência de falhas em um único processo, o problema do consenso não tem solução. Para contornar esta impossibilidade foram desenvolvidas diversas técnicas, que de uma maneira geral, enfraquecem as propriedades do consenso ou inserem limites temporais dinâmicos nos sistemas assíncronos [28].

A técnica mais para contornar a impossibilidade de FLP é baseada na implementação de oráculos para orientar as aplicações quanto ao estado dos processos do sistema. Estes oráculos são módulos distribuídos, chamados de detectores de falhas (*failure detectors*), desenvolvidos por Chandra e Toueg [11, 40], para encapsular as características temporais inseridas ao sistema assíncrono, quando necessário. Com a utilização dos detectores de falhas, as aplicações construídas para ambientes assíncronos podem ser desenvolvidas de maneira mais simples, como se fossem projetadas para ambientes síncronos. Para isto, isenta-se as aplicações de fazer asserções temporais sobre o comportamento do sistema para a detecção da falha, deixando esta tarefa para os detectores de falhas.

Os detectores de falhas são componentes distribuídos, para os quais um processo pode fazer requisições para saber o estado de determinado processo do sistema. Cada processo possui um módulo que fornece as informações sobre o estado dos processos do sistema. Para determinar se um processo é suspeito ou não de estar falho, os detectores fazem asserções temporais sobre o sistema em que estão inseridos. Por esta razão, as informações fornecidas pelo detector são suspeitas sobre o estado dos processos. Em outras palavras, os detectores suspeitam, em determinado momento, que um processo está falho ou correto. Os detectores não sã totalmente confiáveis, pois em determinado momento, um detector pode suspeitar que um processo correto esteja falho ou que um processo falho esteja correto.

Finalmente, existem várias classes de detectores de falhas não confiáveis, classificados de acordo propriedades que definem a confiabilidade da informação que fornecem [11]

aos processos. Cada aplicação deve utilizar uma classe de detectores adequada as suas necessidades. Por exemplo, aplicações com fortes restrições de consistência devem utilizar uma implementação de detectores de falhas que forneça garantias fortes de corretude.

As próximas seções descrevem a difusão confiável e a difusão atômica de eventos em um sistema, bem como o protocolo de confirmação atômica, que são algumas das aplicações do problema do consenso.

### 3.2.1 Difusão Confiável e Ordenação de Mensagens

No consenso os processos precisam decidir por um valor ou ação comuns, enquanto que na difusão confiável os processos precisam entrar em acordo sobre a entrega de um conjunto de mensagens. Informalmente, a difusão confiável é a garantia de que um conjunto de mensagens enviadas aos processos será entregue para todos, ou então não será entregue.

A difusão confiável deve garantir as seguintes propriedades [17]:

- Validade Uniforme: se um processo *entrega* uma mensagem $m$, então algum processo enviou esta mensagem.
- Acordo Uniforme: se um processo *entrega* uma mensagem $m$, então todos os processos corretos irão *entregar* a mensagem $m$.
- Integridade Uniforme: para qualquer mensagem $m$, todo processo *entrega* $m$ no máximo uma vez.
- Terminação: se um processo corretos envia uma mensagem $m$, então todos os processos corretos devem *entregar* $m$.

Protocolos com estas propriedades garantem a entrega das mensagens por todos os processos, mas não garantem a ordem na qual estas mensagens serão entregues por cada um. Em outras palavras, as mensagens poderão ser entregues fora de ordem, a todos os processos. Por exemplo, pode ocorrer que um processo $p$ entregue a mensagem $m$ e depois entregue $m'$, enquanto outro processo $q$ entregue $m'$ e depois entregue $m$. Para

que a ordem de entrega das mensagens seja comum entre todos os processos do sistema, é necessário incluir alguma propriedade de ordenação no protocolo de difusão confiável.

Protocolos de difusão *atômica* garantem as propriedades da difusão confiável e a propriedade de *ordem total*: se dois processos $p$ e $q$ *entregam* as mensagens $m$ e $m'$, então $p$ irá *entregar* $m$ antes de $m'$, se e somente se, $q$ *entregar* $m$ antes de $m'$. Por este motivo, a difusao atômica também é conhecida por *Total Order Broadcast*.

As propriedades apresentadas acima são ditas uniformes, ou seja, se aplicam tanto a processos corretos como falhos. Por exemplo, com um algoritmo de *Uniform Total Order Broadcast*, não é permitido que uma mensagem seja entregue fora de ordem, mesmo que o processo esteja falho. Por outro lado, algoritmos com propriedades que se aplicam apenas a processos corretos são chamados de não uniformes. Apesar de que algoritmos com propriedades não uniformes demandam menor esforço para serem implementados, propriedades não uniformes caracterizam um enfraquecimento de garantias, o que pode levar a inconsistências em aplicações que utilizam algoritmos construídos a partir destas propriedades. As propriedades não uniformes são apresentadas abaixo.

- Validade: se um processo *entrega* uma mensagem $m$, então algum processo correto enviou esta mensagem.
- Acordo: se um processo correto *entrega* uma mensagem $m$, então todos os processos corretos irão *entregar* a mensagem $m$.
- Integridade: para qualquer mensagem $m$, todo processo correto *entrega* $m$ no máximo uma vez.
- Terminação: se um processo correto envia uma mensagem $m$, então todos os processos corretos devem *entregar* $m$.
- Ordem Total: se dois processos corretos $p$ e $q$ *entregam* as mensagens $m$ e $m'$, então $p$ irá *entregar* $m$ antes de $m'$, se e somente se, $q$ *entregar* $m$ antes de $m'$.

A ordem total está baseada na entrega das mensagens pelos processos. Entretanto, pode haver a necessidade de que a entrega seja baseada na ordem de envio das mensagens.

Os protocolos que garantem a ordenação conforme o envio das mensagens pelos processos incluem a propriedade FIFO (*First In First Out*): se um processo correto envia uma mensagem *m* antes de enviar a mensagem *m'*, então não há processo correto que entregue a mensagem *m'* antes de ter entregue *m*.

Os protocolos mais comuns para a difusão confiável assumem que todos os processos pertencentes ao sistema são conhecidos no início do funcionamento do mesmo. Essa asserção pode não ser possível em sistemas onde processos iniciam e encerram sua participação de maneira dinâmica, como é o caso da maioria de sistemas distribuídos implementados com o uso das tecnologias P2P.

#### 3.2.1.1 Ordenação de Mensagens

Esta seção descreve algoritmos distribuídos que realizam a ordenação de mensagens. Algoritmos deste tipo devem ser considerados em trabalhos futuros.

Segundo uma classificação proposta por Défago et al [17], os algoritmos para realizar a difusão atômica podem ser separados em várias classes, de acordo com o processo responsável pela ordenação, que pode ser o processo fonte da mensagem, o processo destino, ou outro processo chamado de sequenciador [17]. Cada classe de algoritmos pode ainda ser subdividida, como mostra a figura 3.1. Há a subclasse denominada sequenciador fixo e sequenciador móvel para a classe de algoritmos de ordenação baseada em um processo sequenciador; as subclasses de algoritmos baseados em privilégio ou em histórico de comunicação para a classe de algoritmos com ordenação realizada pelos processos fonte da mensagem; e a subclasse de acordo entre os destinatários para a classe de algoritmos com ordenação feita pelos destinatários. A seguir estas classes são descritas brevemente.

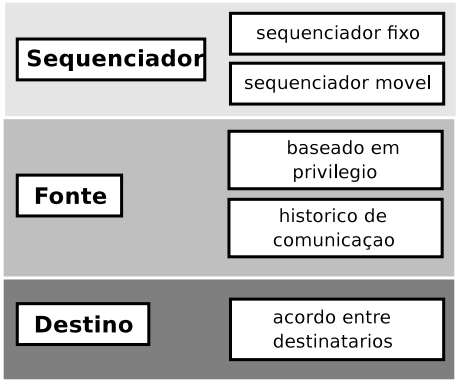


Figura 3.1: Classes de algoritmos para difusão atômica.

A classe de algoritmos baseada em um sequenciador fixo (*fixed sequencer*) é ilustrada na figura 3.2. Nesta classe, um único processo é o responsável por controlar a sequência das mensagens a serem difundidas aos processos, como explicado genericamente a seguir. Um processo que deseja difundir uma mensagem envia esta mensagem ao sequenciador. O sequenciador associa um número de sequência para a mensagem a ser enviada e então faz a difusão desta mensagem, juntamente com a sequência, aos demais processos do grupo. Os processos que recebem as difusões fazem a entrega das mensagens na ordem correta, de acordo com o número de sequência de cada mensagem. Este algoritmo é simples, mas sua desvantagem é que apenas o processo sequenciador mantém as informações sobre a ordenação. O processo sequenciador pode ficar sobrecarregado ou falhar, compromentendo a ordenação.

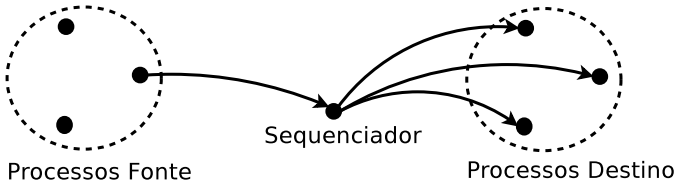


Figura 3.2: Difusão atômica coordenada por um nodo sequenciador.

Além do algoritmo base descrito anteriormente, algoritmos de ordenação com sequenciadores fixos podem ter variações. A primeira variação está no nodo que faz a difusão da

mensagem, que pode ser realizada pelo próprio nodo que deseja enviar a mensagem. O processo que deseja difundir uma mensagem requisita um número de ordem ao processo sequenciador, e faz a difusão da mensagem após ter recebido a resposta. A vantagem desta abordagem é diminuir a carga de trabalho do processo sequenciador, além de acrescentar apenas uma mensagem à quantidade de mensagens que trafegam pela rede. Em contrapartida, o tempo para completar a difusão aumenta, pois é necessário que o processo aguarde a resposta do sequenciador. Em uma outra variação, o processo que deseja enviar uma mensagem faz a difusão desta, e a seguir o nodo sequenciador faz a difusão do número de sequência para a mensagem difundida. Esta variação causa um maior número de mensagens trafegadas pela rede.

Na classe de algoritmos com sequenciadores móveis (*moving sequencer*), a tarefa de controlar a ordem das mensagens é dividida entre um grupo de processos sequenciadores, como ilustrado na figura 3.3. Para realizar a difusão, um processo envia a mensagem para todos os sequenciadores. Os sequenciadores circulam um *token* entre si, juntamente com o número de sequência para a difusão da próxima mensagem. Este *token* define qual sequenciador deve fazer a difusão das mensagens. O sequenciador que possuir o token realiza a difusão da mensagem juntamente com o seu número de sequência. Após certo período de tempo, o token e o último número de sequência são passados para o próximo processo sequenciador.

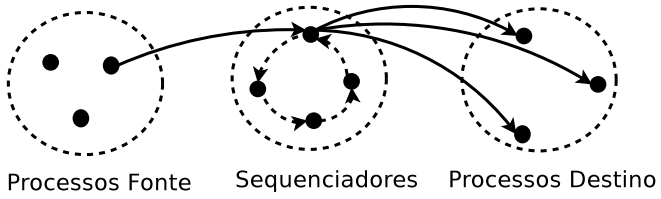


Figura 3.3: Difusão atômica coordenada por um grupo de processos sequenciadores.

Esta abordagem distribui a carga do controle da ordenação entre vários processos. Entretanto, a complexidade de implementação e a latência são maiores, se comparados aos algoritmos com sequenciador fixo.

A classe de algoritmos em que o processo fonte da mensagem faz o controle da ordenação possui uma subclasse de algoritmos baseada em privilégios e outra subclasse baseada no histórico de comunicação.

Nos algoritmos de difusão atômica baseada em privilégio, os processos fonte de mensagens circulam um *token* que permite ao processo que o possui, enviar todas as mensagens que desejar. Esta abordagem, ilustrada na figura 3.4, privilegia processos que tenham muitas mensagens a serem enviadas, enquanto os demais precisam aguardar pelo recebimento do *token*, mesmo que possuam poucas mensagens a serem enviadas.

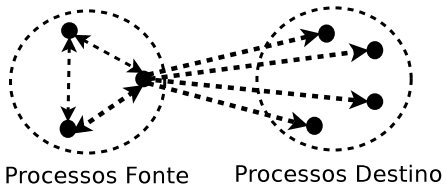


Figura 3.4: Difusão atômica coordenada pelos processos que enviam mensagens.

Na subclasse onde a ordenação é feita através do histórico de comunicação entre os processos, toda mensagem carrega um indicativo lógico ou físico de tempo (*timestamp*), que permite que os processos entreguem as mensagens em ordem, através da observação do *timestamp* das mensagens trafegadas pela rede.

Na classe de algoritmos para difusão atômica coordenada pelos destinatários, mostrada na figura 3.5, a ordenação é realizada pelos processos que entregam as mensagens (destinatários), através de um acordo. O acordo pode ser sobre o número de sequência da mensagem, sobre um conjunto de mensagens que devem ser entregues ou mesmo sobre a ordem na qual devem entregar as mensagens. No caso descrito a seguir, os processos atingem um acordo sobre o número de sequência das mensagens. Os processos destino associam um *timestamp* local a cada mensagem recebida e enviam este valor aos demais processos. Os processos escolhem o maior dentre os valores recebidos para considerar como o *timestamp* da mensagem (*timestamp* global). Os processos entregam a mensagem de acordo com o valor deste *timestamp* global.

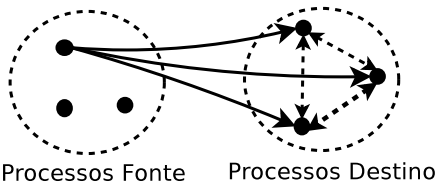


Figura 3.5: Difusão atômica coordenada pelos destinatários.

### 3.2.2 Protocolos de Confirmação

Os protocolos de confirmação são utilizados para garantir a atomicidade das operações ou transações em sistemas distribuídos. Em outras palavras, garantir que uma ação seja realizada integralmente por todos os processos do sistema distribuído, ou então que nenhuma ação seja realizada [5]. Estes protocolos podem ser utilizados para a construção de banco de dados distribuídos, quando da necessidade de certeza da execução de transações em todas as instâncias do banco.

O 2PC (*Two Phase Commit* - protocolo de confirmação em duas fases) é o mais conhecido dentre os protocolos de confirmação e tem sua execução realizada em duas fases [29, 31].Para a execução deste protocolo há um processo *coordenador*, que tem a responsabilidade de definir se a operação deve ser realizada (*commit*) ou abandonada (*abort*). Os outros processos do sistema são chamados de *participantes*.

Na primeira fase, o coordenador envia uma mensagem *commit-request* a todos os participantes. A seguir, os participantes enviam seu voto ao coordenador. O voto de cada participante contém a sua indicação a favor do *commit* ou do *abort* da operação, de acordo com suas condições locais. Na segunda fase, o coordenador considera as respostas de todos os participantes. Se todos responderam *commit*, o coordenador envia uma mensagem *commit* a todos os participantes. Caso contrário, o coordenador envia uma mensagem *abort* a todos os processos participantes. Cada processo participante executa a ação da mensagem recebida do coordenador.

No caso da falha de um processo participante, o coordenador pode detectar esta falha através de um mecanismo de *timeout* e decidir de acordo com as respostas dos outros

processos. No caso de falha do processo coordenador, um participante pode ficar esperando pela mensagem de confirmação da ação. O 2PC é bloqueante se existe falha do coordenador. Uma alternativa para o bloqueio pode ser o participante, após detectar a falha do coordenador através de *timeout*, contactar outros participantes para verificar se a resposta do coordenador foi *commit* ou *abort*. Se outros participantes também não obtiveram resposta do coordenador, a ação a realizar deve ser *abort*. Em sistemas assíncronos, considera-se que canais de comunicação são confiáveis ou têm perdas equitáveis, para permitir as propriedades de *safety* e *liveness* dos algoritmos.

## 3.3 Técnicas de Replicação

A replicação é uma abordagem para aumentar a disponibilidade e aumentar a corretude dos sistemas. Sua principal idéia é inserir redundância, na forma de várias cópias de dados ou processos, para que seja possível acessar o serviço mesmo se ocorrer falhas em algumas destas cópias [28]. As técnicas de replicação exigem controle para garantir que todas as cópias se mantenham consistentes ao longo da execução do sistema. Um detalhe importante é que a replicação deve ser aplicada de maneira transparente ao usuário, para que este utilize o sistema replicado como se não existissem cópias do recurso original.

As duas principais técnicas para a implementação da replicação são a replicação passiva [9] e a replicação ativa [52], descritas adiante. Outras técnicas intermediárias também existem.

Na replicação passiva ou *primary backup* várias réplicas funcionam como *backups* e apenas uma atua como réplica principal, atendendo as requisições de clientes. As ações são realizadas pela réplica principal, enquanto as demais apenas escutam e mantêm o estado consistente com a principal. No caso de uma requisição de consulta, a réplica principal responde imediatamente. No caso de atualização, como ilustrado na figura 3.6, a réplica principal realiza a atualização localmente e envia informações sobre a operação realizada a todas as outras réplicas, através de um protocolo de difusão confiável. Após receber a confirmação de todas as outras réplicas corretas, a principal envia uma confirmação ao processo que fez a requisição.

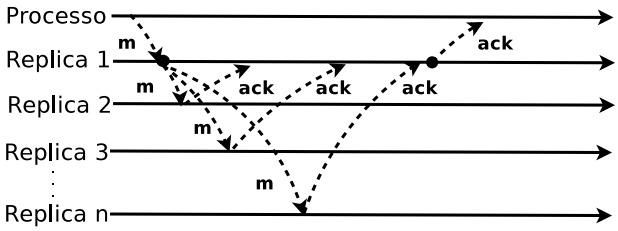


Figura 3.6: Mensagem de atualização em sistema com replicação passiva.

Utilizando difusão confiável para a transmissão de mensagens, todas as réplicas são mantidas consistentes. No caso de falha da réplica principal, algum *backup* assume este papel. A eleição de qual réplica assume as ações da principal pode ser feita utilizando um protocolo de acordo, ou então considerar a réplica com o maior (menor) identificador. Em ambos os casos, todas as réplicas devem conhecer o conjunto completo de réplicas correta. Para isto, podesm ser utilizados serviços de gestão de grupos, que serão descritos na próxima seção. A desvantagem desta forma de replicação é a parada temporária no serviço durante a troca da réplica principal falha.

Na estratégia de replicação ativa [52], ilustrada na figura 3.7, todas as réplicas executam as mesmas ações e respondem a todas as requisições. Desta forma, um processo recebe várias respostas para cada requisição realizada. Possuindo várias respostas, o processo pode executar algum procedimento para validá-las. A validação das respostas permite inclusive desconsiderar resultados inconsistentes, que podem ter sido gerados a partir de réplicas sob a ação de falhas bizantinas. Uma vantagem da replicação ativa é abranger modelos que permitem falhas bizantinas, entretanto, com a necessidade de um maior número de mensagens transmitidas. Assim como na replicação passiva, a transmissão das mensagens deve ser feita através de um protocolo de difusão atômica, para manter a consistência do sistema.

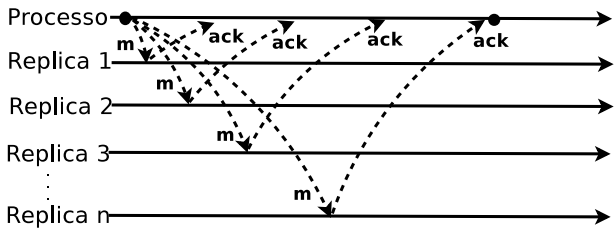


Figura 3.7: Mensagem de atualização em sistema com replicação ativa.

Na replicação passiva, quando ocorre particionamento do sistema, apenas uma partição contém a réplica principal (primária). Uma opção para tratar o particionamento é fazer com que as demais partições não elejam uma réplica principal. Desta forma, as requisições advindas de processos destas outras partições não obtêm respostas, causando uma diminuição da disponibilidade do sistema. Esta forma de replicação pode ser aplicada a sistemas que possuem restrições fortes quanto à consistência dos dados [28]. Outra opção é permitir que todas as partições contenham uma réplica principal, que possa responder requisições, ao custo de possíveis inconsistências entre os subsistemas formados pelas diferentes partições. Uma maneira de diminuir a ocorrência de inconsistência é permitir que apenas uma partição continue realizando requisições de atualização, enquanto nas demais partições apenas as requisições de consulta são respondidas.

No caso de particionamento da rede na replicação ativa, ocorre caso semelhante ao da replicação passiva, onde todas as partições mantêm certo nível de serviço. Neste caso, todas as réplicas estão automaticamente programadas para responder todas as requisições, independentemente de qual partição estejam.

Para resolver os problemas de particionamento em sistemas replicados, surgiram duas linhas de pesquisa, uma considerada otimista e a outra pessimista [31]. A linha otimista considera que as operações realizadas em cada partição não serão conflitantes, permitindo que as réplicas destas partições tenham seu conteúdo fundido quando da recuperação do canal. A pessimista considera que as operações realizadas em cada partição provavelmente serão conflitantes, então deve-se ter meios para evitar conflitos quando da recuperação da falha do canal de comunicação.

## 3.4 Serviço de Gestão de Grupos

Para manter um grupo formado por processos, é fundamental que se tenha mecanismos para permitir entradas e saídas de processos do grupo, e fornecer informações sobre quem são e quais as características dos seus membros. Esse serviço é chamado de *serviço de gestão de grupos* [30, 13]. O serviço de gestão de grupos pode ser empregado na solução de diversos problemas de sistemas distribuídos confiáveis.

A composição do grupo e a comunicação entre os processos deste grupo devem ser realizadas de forma transparente. Pode-se visualizar o serviço de gestão de grupos como um *middleware*, inserido abaixo da camada de aplicação que utiliza os serviços do grupo [28]. A figura 3.8 mostra um exemplo de arquitetura para um serviço de gestão de grupos []. Em outras palavras, a aplicação que utiliza o *serviço de gestão de grupos* deve ver e acessar o grupo como se existisse uma única entidade. O acesso deve ser realizado através de primitivas fornecidas pelo serviço de gestão de grupos.

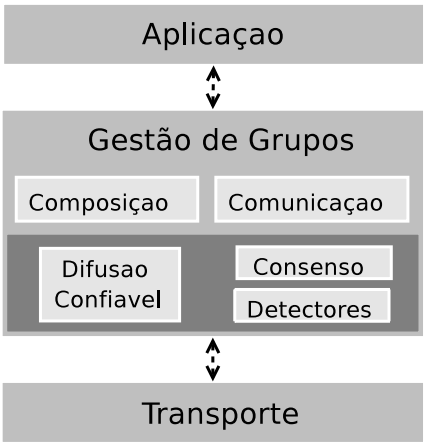


Figura 3.8: Exemplo da estrutura de um serviço de gestão de grupos.

O serviço de gestão de grupos em grupos é formado principalmente pelos componentes de *comunicação* e de *composição*. O primeiro componente é responsável por toda a comunicação entre os membros do grupo, enquanto o segundo componente realiza a manutenção da composição do grupo. O componente de comunicação utiliza protocolos de difusão confiável e de confirmação atômica para realizar a troca de mensagens entre os

membros do grupo. Este componente também controla a comunicação entre os processos que pertencem ao grupo e os processos que não pertencem ao grupo, de forma a permitir a utilização do sistema e atualização dos serviços. O componente de manutenção da composição de grupos deve fornecer aos membros deste, informações consistentes sobre o estado do grupo. Essas informações são principalmente sobre o conjunto de processos ativos no grupo.

O serviço de gestão de grupos normalmente envolve consenso entre participantes, utilizando-se de protocolos de acordo para ser implementado. Em ambientes assíncronos, deve-se ter primitivas que permitam detectar falhas com a confiabilidade necessária para os serviços do grupo.

Muito foi desenvolvido na área de serviço de gestão de grupos desde os anos 90. Entre os trabalhos significativos desta área, vários geraram plataformas, exemplos incluem os sistema Isis [6], Horus [49] e JGroups [2]. Na sua maioria, estas plataformas são construídas seguindo um modelo de camadas, formando pilhas de protocolos. Cada camada se comunica com a inferior para obter serviços, e se comunica com a camada superior para fornecer serviços.

Um exemplo de pilha de protocolos da camada localizada entre a camada de transporte e a aplicação pode incluir uma camada de detecção de falhas mais abaixo, uma camada de protocolos de difusão confiável acima desta e por último a camada do serviço de gestão de grupos. Apenas esta última camada é vista pela camada de aplicação.

### 3.4.1 Grupos Particionáveis

Quando o sistema sobre o qual o serviço de gestão de grupos é executado pode se particionar, são necessárias técnicas específicas, algumas das quais descritas a seguir [28]: serviço de gestão de grupos com componente primária e com componentes particionáveis. O serviço de gestão de grupos com componente primária permite a existência de uma única visão do grupo a todo momento, de maneira mais simples que com componentes particionáveis. No caso de um serviço com componente principal, caso haja o particionamento do grupo devido a falhas, apenas o subgrupo majoritário ou componente principal

continua a prover o serviço. O componente principal é aquele que contêm a maioria dos processos corretos, após o particionamento. Esta forma de gestão faz com que, a todo momento, a visão do grupo disponível seja apenas a do componente principal.

A utilização de componentes particionáveis propicia um enfraquecimento das propriedades de consistência, pois as várias partições podem ter visões com pequenas inconsistências entre si. No caso de particionamentos, cada componente age de maneira independente. Os processos comunicam-se como se não houvesse outro subgrupo de processos, dos quais estão isolados. Apesar do enfraquecimento das propriedades de consistência, esta abordagem permite oferecer o serviço completo ou uma parte, a partir de qualquer partição do sistema, aumentando a disponibilidade. Quando a falha no canal de comunicação é restaurada, é necessário realizar algoritmos de reconciliação (*group merging*) para fazer com que o estado global do sistema se torne novamente consistente.

As diferentes técnicas para tratar particionamentos podem ser aplicadas a sistemas também diferentes. Em sistemas nos quais a ocorrência de falhas é pequena ou a necessidade de consistência é grande, pode-se utilizar a composição com componente primária. Em sistemas em que há grande ocorrência de particionamento do sistema, a melhor opção pode ser a utilização de componentes particionáveis [28].

### 3.4.2 Ferramentas de Comunicação em Grupos

Diversas ferramentas para gestão de grupos têm sido desenvolvidas para possibilitar a construção de sistemas distribuídos confiáveis. Dentre estas ferramentas, o JGroups [2], Pastry e Scribe [10], e o Spread [56] são brevemente descritas adiante. O JXTA-RM (*JXTA-Reliable Multicast*) [34] é um sistema que implementa multicast confiável na plataforma JXTA, e também é descrito a seguir.

O Pastry [50] é um sistema formado por uma rede *overlay* de nodos conectados à Internet. Este sistema possui uma estrutura própria para a formação da rede P2P, busca de objetos e roteamento. Segundo Rowstron e Druschel [50], as principais características do Pastry é ser decentralizado, tolerante a falhas e escalável. Cada nodo possui um identificador único dentro deste sistema, e é responsável por responder ou encaminhar

requisições de outros nodos. Todos os nodos mantém informações locais sobre nodos vizinhos. O Pastry é responsável por disseminar informações de saída ou entrada de nodos da rede, aos demais nodos e às aplicações que o utiliza. Dentre as aplicações construídas a partir da estrutura do Pastry, há o Scribe, descrito a seguir.

O Scribe [10] é um sistema para multicast de mensagens, com propósito de fornecer um serviço confiável e escalável para aplicações de gestão de grupos. O Scribe pode ser executado em grande quantidade de processos por grupo, e grande quantidade de grupos no sistema todo, mantendo a eficiência, mesmo com altas taxas de entrada e saída de nodos. Qualquer nodo do Scribe pode criar um grupo, no qual outros nodos podem participar. Cada grupo também possui um identificador único dentro do sistema Scribe, fornecido pela estrutura do Pastry. Nodos podem criar, participar e enviar mensagens para vários grupos. O Scribe não especifica a ordem de entrega das mensagens dentro de um grupo. Cabe à aplicação que o utiliza definir a ordem de entrega das mensagens, de acordo com sua necessidade. O Scribe utiliza o Pastry para gerenciar a criação de grupos, a entrada e saída de membros e para construir uma árvore de disseminação para as mensagens. Sua estrutura é totalmente descentralizada e as decisões de roteamento que formam a árvore de disseminação são baseadas em informações locais que os nodos mantém sobre seus vizinhos na rede.

O JGroups [2] é formado por um conjunto de ferramentas desenvolvidas em Java, pela Universidade de Cornell, cuja idéia é justamente não ser totalmente transparente, de forma que o desenvolvedor da aplicação que o utiliza, tenha liberdade para definir suas estratégias de confiabilidade. O principal objetivo do JGroups é estabelecer uma biblioteca com padrões de estruturas e algoritmos frequentemente utilizados, para facilitar a construção de aplicações que utilizem um serviço de gestão de grupos. Sua estrutura tem a forma de uma pilha de protocolos, que foram desenvolvidos em classes ou módulos. Desta forma, os desenvolvedores podem escolher quais módulos utilizar para construir uma aplicação. A abordadem da gestão de grupos pode ser utilizada para aumentar a disponibilidade de aplicações, para balancear a carga de trabalho entre processos servidores, ou mesmo para dividir uma tarefa em tarefas menores, a serem executadas pelos vários processos

que formam o grupo. Por estas razões, o JGroups utiliza-se da gestão de grupos com qualidade de serviço, na forma de garantias na entrega e na ordem em que as mensagens serão entregues pelos membros do grupo. As diferentes garantias de entrega e ordenação podem ser escolhidas de acordo com a necessidade da aplicação. A principal abstração do JGroups é um canal virtual, pelo qual os processos podem se conectar a um grupo, através do identificador deste grupo. Depois de conectados, os processos podem enviar e receber mensagens, e também notificações de entrada e saída de membros no grupo.

O Spread [56] é uma ferramenta de gestão de grupos que provê uma estrutura confiável para a comunicação entre processos, desenvolvido pelo Centro de Redes e Sistemas Distribuídos da Universidade Johns Hopkins. O Spread funciona como um canal de comunicação entre processos, para ser utilizado por aplicações distribuídas. Este canal fornece desde a difusão confiável e a garantia de entrega com ordenação total de mensagens, até a ordenação FIFO, requerida para a transmissão de vídeo e áudio (*streaming*). Estas propriedades são garantidas mesmo na presença de falhas de processos ou particionamento da rede. Outras características do Spread incluem simplicidade de uso e escalabilidade, pois sua estrutura suporta uma grande quantidade de grupos. O Spread tem como foco principal a comunicação sobre redes de longa distância.

O JXTA-RM [34] é um protocolo para realizar comunicação confiável (*reliable multicast*) entre *peers* do sistema JXTA. Este protocolo foi construído utilizando componentes da linguagem JAVA e abstrações de canais de comunicação do JXTA. A interface inclui um método que possibilita que um *peer* fique escutando o canal para o recebimento de mensagens, e um método para difusão confiável de mensagens a um grupo de *peers*, denominado método *propagate*. O método *propagate* foi implementado de duas maneiras, descritas a seguir. Uma das implementações do *propagate* utiliza o protocolo de *Rendezvous* estabelecido pelo JXTA, para conseguir passar por roteadores e firewalls e chegar a todos os membros do grupo, mesmo que estejam em redes físicas diferentes. A outra implementação não utiliza o protocolo de *Rendezvous*, possuindo garantias de entrega das mensagens apenas para os *peers* que estão na mesma LAN. Para redes JXTA com *peers* localizados fisicamente na mesma LAN, esta implementação é a mais eficiente.

O JXTA-RM foi implementado através de três protocolos de multicast fornecidos pela biblioteca JRMS (*Java Reliable Multicast Service*) do JAVA: UM, TRAM e LRMP, descritos brevemente a seguir. O UM (*Unreliable Multicast*) é um protocolo que não fornece controle sobre os dados transmitidos. O TRAM (*Tree Based Reliable Multicast Protocol*) foi desenvolvido para suportar transmissão de dados confiável a partir de um único *peer* fonte para múltiplos destinos. Este protocolo é escalável para um grande número de destinos, sem que o *peer* fonte seja inundado com mensagens de notificação de recebimento. O LRMP (*Ligtweight Reliable Multicast Protocol*) funciona em ambientes de rede heterogêneos e suporta envio confiável de mensagens por múltiplos processos de maneira concorrente. Para a utilização do JXTA-RM pode-se escolher qual dos protocolos descritos anteriormente será utilizado para o envio das mensagens.

# Capítulo 4

# A Ferramenta Proposta Para Comunicação Confiável em Sistemas P2P

Este capítulo apresenta uma ferramenta para a construção de sistemas P2P robustos. Seu principal objetivo é permitir que as aplicações apresentem confiabilidade, de maneira transparente ao usuário. Em outras palavras: aplicações confiáveis têm a mesma interface e funcionalidade de aplicações tradicionais. A ferramenta apresenta as funcionalidades básicas do serviço de gestão de grupos no contexto de sistemas P2P, implementando também protocolos de confirmação e eleição de líder. Uma aplicação que permite o download confiável de arquivos foi construída como exemplo de funcionamento da ferramenta. Esta aplicação foi implementada utilizando a plataforma JXTA, que fornece uma estrutura pré definida para a criação de sistemas P2P, descrita no próximo capítulo.

Este capítulo está organizado da seguinte forma: a seção 4.1 apresenta a ferramenta e sua estrutura, a seção 4.2 mostra o módulo gestão de grupos, a seção 4.3 o protocolo de confirmação e a seção 4.4 detalha o módulo de eleição.

## 4.1 Arquitetura da Ferramenta

A ferramenta apresentada neste trabalho foi construída em módulos, que interagem para prover a funcionalidade necessária para a camada de aplicação. A figura 4.1 mostra uma representação desta estrutura. Os principais módulos que fazem parte da ferramenta são os do serviço de gestão de grupos, protocolo de confirmação e eleição, descritos ao longo deste capítulo.

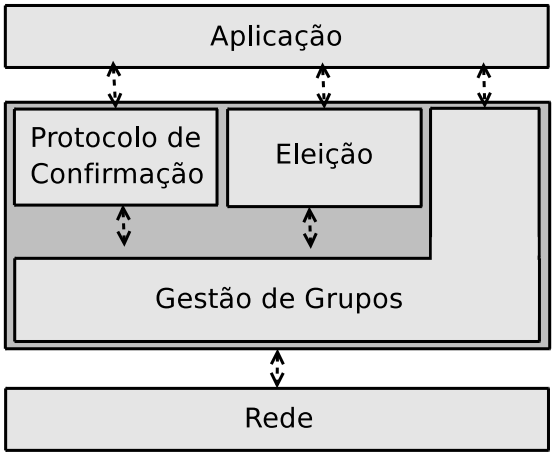


Figura 4.1: Estrutura em módulos da ferramenta.

A ferramenta utiliza a rede para criar um canal de comunicação entre os *peers* que desejam fazer parte de um grupo. Acima desta rede, forma-se uma rede P2P virtual e independente. Considera-se que a topologia da rede é completa, e que cada nodo de um grupo de nodos consegue se comunicar diretamente com todos os outros.

Considera-se um sistema assíncrono, ou seja, não são feitas hipóteses temporais sobre a realização das ações efetuadas pelos processos ou pelos canais. Um processo pode falhar por parada (*crash*), através de um colapso brusco ou saída deliberada do sistema. São considerados canais de comunicação com perdas equitáveis (*fair-lossy*). Neste caso, se um processo $p$ emite uma mensagem $m$ a um processo $q$ um número infinito de vezes e dado que o processo $q$ não falhe, então $q$ recebe $m$ de maneira definitiva.

O módulo de gestão de grupos mantém um grupo de processos, também chamados de *peers*, e provê a comunicação entre *peers* que pertencem ao grupo, e destes com *pe-*

*ers* externos. O serviço de gestão de grupos também é responsável por detectar *peers* falhos dentro do grupo e manter uma lista de membros sem-falha, utilizada pelos demais módulos. Este módulo é de execução obrigatória em todos os *peers* que fazem parte do grupo.

O módulo do protocolo de confirmação é o responsável pela realização de ações atômicas dentro do grupo. Esta funcionalidade pode ser utilizada em aplicações P2P para trabalho colaborativo, onde *peers* podem alterar o conteúdo de arquivos simultaneamente. Este módulo recebe as mensagens do protocolo de confirmação enviadas por membros do grupo, executa as fases do protocolo e age de acordo com a decisão tomada através deste protocolo. As mensagens são enviadas utilizando o canal de comunicação mantido pelo serviço de gestão de grupos.

O módulo de eleição de líder é utilizado para definir um *peer* para tratar requisições recebidas tanto de *peers* internos quanto externos ao grupo. Este módulo pode ser utilizado para eleição de *peers* para realizar tarefas, como por exemplo a eleição de um *peer* de um grupo para enviar um arquivo a outro *peer*. Como o protocolo de confirmação, o módulo de eleição também é opcional, e utiliza as funções de envio e recebimento de mensagens fornecidas pela gestão de grupos.

As funcionalidades dos módulos da ferramenta são utilizados pela camada de aplicação conforme a necessidade dos serviços que esta deve oferecer aos seus usuários.

## 4.2 Serviço de Gestão de Grupos

O serviço de gestão de grupos proposto utiliza a estrutura de rede *overlay* do JXTA para criar um canal de comunicação entre os *peers* que desejam fazer parte de um grupo. Considera-se que a topologia da rede é completa, e que cada nó de um grupo consegue se comunicar diretamente com todos os outros.

O serviço de gestão de grupos mantém um grupo de *peers* e provê a comunicação entre *peers* que pertencem ao grupo e destes com *peers* externos. A gestão de grupos, através do seu componente de *group membership*, também é responsável por detectar *peers* falhos dentro do grupo, manter uma lista de membros sem-falha, chamada de *visão*

*do grupo* e fornecer mecanismos para a entrada e saída de membros no grupo [13, 30]. Na próxima sub-seção, apresentamos os protocolos e funcionalidades deste componente, que estão ilustrados na figura 4.2.

### 4.2.1 Serviço de Gestão da Composição do Grupo

**Entrada no Grupo** Ao ser inicializado, um *peer* executa uma função de entrada no grupo, informando qual a identificação deste grupo. Este *peer* envia uma mensagem inicial ao grupo, solicitando a sua inscrição. Se não receber resposta, cria o grupo, pois deduz que não há outros membros ainda. Caso contrário, insere a identificação dos *peers* dos quais recebeu resposta na sua visão local. Os *peers i* que receberam uma mensagem inicial de algum *peer j*, inserem o *peer j* em sua visão e enviam uma mensagem de confirmação ao *peer i*.

**Atualização das Visões** O serviço de gestão de grupos possui uma função que faz o recebimento e tratamento das suas mensagens para manutenção das visões do grupo. Toda vez que uma mensagem é recebida, o *peer* que a recebeu insere a identificação do *peer* que a enviou em sua visão local do grupo. Com exceção da mensagem inicial (de entrada no grupo), todas as demais mensagens trocadas entre os *peers* carregam a visão local do *peer* que enviou a mensagem. Todos os *peers* do grupo armazenam a visão dos outros *peers* do grupo para construir uma visão única, que será descrita adiante. Além disso, a visão única que um *peer* mantém do grupo contém a identificação dos *peers* de maneira ordenada lexicograficamente pela identificação dos membros do grupo.

A cada intervalo de tempo, cada *peer* envia mensagens a todo o grupo, para avisar que está sem-falha (*Estou Vivo*) e para enviar sua visão local atual. Um *peer* também envia a sua visão local a todo o grupo toda vez que esta visão é alterada pela detecção de entrada, saída ou falhas de membros.

**Visão de Grupo** A visão de grupo representa o grupo para a aplicação. Ela deveria ser única para todos os membros que compõem o grupo. Entretanto, devido à alta dinamicidade dos sistemas P2P, caracterizada por entradas e saídas frequentes, e ao

```
/* Variáveis: */
  ListaDePeers visaoDeGrupo = null
  ListaDePeers visaoLocal = null
  Lista de ListaDePeers lVisoes = null

|| EntraGrupo(IdGrupo):
  Pesquisa pelo grupo IdGrupo
  Se encontrou
    Envia mensagem inicial aos membros do grupo IdGrupo
  Senão
    Cria grupo IdGrupo

|| RecebeMensagensComunGrupo(mensagem recebida: msgComunGrupo):
/* Quando o peer i recebe uma mensagem de um peer j */
  Caso msgComunGrupo seja:
    Mensagem inicial recebida do peer j:
      Envia resposta e visaoLocal ao peer j
      Insere peer j na visaoLocal
    Mensagem de resposta da mensagem inicial
      Insere peer j na visaoLocal
      Armazena visão do peer j em lVisoes
    Mensagem "Estou Vivo"
      Insere peer j na visaoLocal
      Armazena visão do peer j em lVisoes

|| AtualizaVisao():
/* Peer i executa periodicamente: a cada intervalo de tempo */
  Se peer i não recebeu mensagem de algum peer j remoto de visaoLocal
    Exclui peer j de visaoLocal
/* Executa a cada alteração em visaoLocal e a cada intervalo de tempo */
  Envia mensagem "Estou Vivo" com visaoLocal aos membros do grupo
  visaoDeGrupo = peers em todas as visões de lVisoes
```

Figura 4.2: Serviço de Gestão da Composição do Grupo

assincronismo, o que impede que falhas sejam detectadas de forma segura, a obtenção de tal unicidade ou concordância na visão dos membros do grupo torna-se um tarefa de complexidade não trivial. Uma estratégia para contornar tal complexidade consiste em desenvolver protocolos simples, que suportam dinamismo e crescimento em larga escala, porém às custas do oferecimento de serviços de grupo com garantias mais fracas de consistência [28]. O protocolo aqui apresentado segue essa estratégia.

Um *peer* decide *unilateralmente* quais *peers* fazem parte da sua visão única, também chamada de visão de grupo, a partir das visões locais recebidas dos demais *peers* do sistema. Assim, fazem parte da visão de grupo do *peer i*, todos os *peers j* que estão contidos em todas as visões locais recebidas pelo *peer i*. Ou seja, todos os *peers j* que não foram considerados falhos por todos aqueles que fazem parte da visão local do *peer i*. Este conjunto será representado pela interseção das visões locais recebidas pelo *peer i*.

Como o sistema é assíncrono, um *peer* lento pode ser considerado falho na visão de grupo. Caso um *peer* receba uma visão local da qual não participa, executa novamente o procedimento de inicialização. Um *peer* pode receber uma visão da qual não participa, caso outro membro do grupo o tenha detectado como falho. A cada intervalo de tempo, cada *peer* verifica se mensagens foram recebidas dos *peers* remotos que estão em sua visão local. Caso mensagens de um *peer* não sejam recebidas no período, exclui este *peer* de sua visão local.

Observe que o protocolo apresentado permite que visões de grupo distintas possam coexistir no sistema em determinado momento. Entretanto, se durante um certo período de tempo o sistema se mantém estável (sem entradas e saídas frequentes) e além disso, não há suspeitas indevidas de falhas, o protocolo apresentado permite com que as diversas visões de grupo locais convirjam para uma única visão. Como os sistemas atravessam períodos de estabilidade frequentes, esta convergência será o caso mais comum, na prática. Sendo assim, a visão de grupo conterá todos os *peers* sem-falha presentes no sistema.

O serviço de gestão de grupos disponibiliza uma interface com funcionalidades úteis à aplicação executada pelos *peers*, como mostrado na figura 4.3 e descrito abaixo. Estas funcionalidades incluem a consulta à visão de grupo do *peer*, o recebimento de mensagens

```
|| ObtemVisaoGrupo():
/* Funcão para consulta 'a visão do grupo */
  Retorna visão do grupo armazenada localmente

|| EnviaMsgGrupo(msg, idPeer):
/* Funcão de envio de mensagens ao grupo ou a um peer especifico */
  Se (idPeer contem valor)
    Envia mensagem msg ao peer idPeer
  Senão
    Envia mensagem msg ao grupo

|| RecebeMsgGrupo():
/* Funcão de recebimento de mensagens enviadas ao grupo */
/* peer i recebeu, peer j enviou */
  Quando chegar mensagem msg
  Se msg pertence ao modulo comunicacão em grupo
    Avisa modulo de comunicacão em grupo sobre nova mensagem
  Se msg pertence ao modulo do protocolo de confirmacão
    Avisa modulo do protocolo de confirmacão sobre nova mensagem
  Se msg pertence ao modulo eleicão de lider
    Avisa modulo de eleicão de lider sobre nova mensagem
```

Figura 4.3: Interface da comunicação em grupos.

enviadas aos membros do grupo e o envio de mensagens ao grupo todo ou a um *peer* específico, membro ou não do grupo, citadas a seguir. A função *ObtemVisaoGrupo()* retorna a visão atual do grupo à quem a utiliza. A função *EnviaMsgGrupo(msg, idPeer)* envia mensagens ao grupo todo ou a um *peer* específico, membro ou não do grupo. No caso de envio a um *peer* específico, a sua identificação (*idPeer*) precisa ser informada. Finalmente, a função *RecebeMsgGrupo()* recebe e classifica as mensagens de acordo com seu destino: o módulo de comunicação em grupos, o módulo do protocolo de confirmação ou o módulo de eleição de líder.

## 4.3 Protocolo de Confirmação: *Two Phase Commit*

O módulo do protocolo de confirmação implementa o protocolo *Two-Phase Commit* (2PC). Este protocolo é responsável pela execução de ações atômicas por todos os mem-

bros do grupo. Para a execução desta ação, um *peer* do grupo age como *coordenador*, enquanto os demais membros agem como *participantes*.

O *peer* coordenador envia uma mensagem contendo a ação, utilizando o módulo do protocolo de confirmação, espera as respostas dos participantes, decide entre confirmar ou cancelar a mensagem de acordo com as respostas e envia esta confirmação novamente aos participantes. Os participantes recebem a mensagem, enviam uma resposta ao coordenador e aguardam uma nova mensagem do coordenador, com a confirmação para a mensagem recebida previamente. Caso o coordenador decida por confirmar a ação, todos os membros do grupo *entregam* a mensagem, ou seja, executam a ação contida na mensagem recebida. Caso contrário, a mensagem recebida é ignorada por todos os *peers* do grupo. Os passos do protocolo de confirmação serão detalhados adiante.

Um *peer* envia uma mensagem através do protocolo 2PC utilizando a função *enviaMsg2PC()*, descrita na figura 4.5, e recebe mensagens deste protocolo utilizando as funções mostradas nas figuras 4.6 e 4.7. Ao enviar uma mensagem em 2PC, um *peer* apenas poderá enviar uma outra mensagem em 2PC após enviar a confirmação da primeira mensagem enviada. Isto não impede o recebimento de mensagens vindas de outros *peers* do grupo. Para executar o protocolo de confirmação, este módulo utiliza as funcionalidades de envio e recebimento de mensagens do módulo de comunicação em grupos.

O módulo do protocolo de confirmação utiliza algumas variáveis para indicar o estado de *peer*. Estas variáveis indicam se um *peer* age como coordenador ou participante de uma ação atômica, em determinado momento. Estas variáveis são mostradas na figura 4.4: *enviandoMsg2PC*, indica que este *peer* é coordenador de uma mensagem que ainda não foi confirmada; *esperandoConfirmacao*, indica que um *peer i* está agindo como participante na execução do protocolo de confirmação para uma mensagem e está aguardando a confirmação; *mensagem2PCEsperando*, é a mensagem 2PC recebida do coordenador e armazenada pelo participante, da qual o participante está esperando a confirmação; *msg2PCEnviada*, é a mensagem 2PC enviada e armazenada pelo *peer* coordenador, que ainda não foi confirmada.

```
/* Variaveis: */
  /* Indica que este peer enviou uma msg 2PC que ainda nao enviou confirmacao */
  boolean enviandoMsg2PC = false
  /* Indica que este peer recebeu uma mensagem 2PC e ainda nao foi confirmada */
  boolean esperandoConfirmacao = false
  /* Mensagem 2PC recebida por este peer e ainda nao confirmada */
  mensagem2PC mensagem2PCEsperando = null
  /* Mensagem 2PC enviada por este peer e ainda nao confirmada */
  mensagem2PC msg2PCEnviada = null
```

Figura 4.4: Variáveis do protocolo de confirmação.

## Protocolo de Confirmação: Coordenador

A execução do protocolo de confirmação pelos *peers* do grupo é realizada em duas fases. O algoritmo deste protocolo é mostrado nas figuras 4.5, 4.6 e 4.7 e descrito a seguir. Na primeira fase do protocolo de confirmação, o *peer* coordenador envia a mensagem a todos os membros do grupo e aguarda as respostas para a mensagem enviada, como mostrado na figura 4.5. Esta mensagem é armazenada em uma lista, para que possa ser consultada posteriormente pelo coordenador. O coordenador de uma mensagem é o próprio *peer* que deseja enviar esta mensagem.

A segunda fase do algoritmo envolve a análise das respostas dos participantes e envio da confirmação. Quando as respostas de todos os *peers* participantes forem recebidas, o coordenador faz a análise, conforme pseudo-código mostrado na figura 4.6. As respostas recebidas para a mensagem podem ser *cancel* ou *confirm*. Se todas as respostas recebidas são *confirm*, a decisão do coordenador será confirmar a mensagem. Se há pelo menos uma resposta *cancel*, a decisão será cancelar a mensagem enviada previamente. O passo final do protocolo, para o coordenador, é o envio da confirmação da mensagem 2PC a todos os membros do grupo. Esta confirmação contém a decisão feita pelo coordenador.

```
|| EnviaMsg2PC(mensagem2PC: msg2PC):
/* Para enviar uma mensagem 2PC */
/* Esta função e' executada paralelamente ao recebimento de mensagens */
  Repete enquanto não enviar mensagem 2PC ao grupo:
    Se esta esperando confirmação de mensagem recebida
      dorme
    Senao
      Envia mensagem 2PC ao grupo
      decisao = EsperaRespostas(msg2PC)
      Envia mensagem com decisão e visão do peer i ao grupo
      Se decisao for confirmar
        Avisa aplicação que mensagem foi confirmada
      Senao
        Avisa aplicação que mensagem foi cancelada


|| EsperaRespostas(mensagem2PC: msg2PC):
/* Espera respostas do grupo para mensagem 2PC enviada */
  Espera mensagem de todos os membros do grupo
  Se todas as respostas são confirmação
    respostas = confirmar
  Senao
    respostas = cancelar
  retorna respostas
```

Figura 4.5: Algoritmo do envio de mensagens do protocolo de confirmação executado pelo coordenador.

## Protocolo de Confirmação: Participantes

Do lado dos participantes, a primeira fase é o envio da resposta ao *peer* coordenador e espera pela confirmação, e a segunda fase envolve o recebimento da confirmação e a execução de acordo com a decisão do coordenador. As funções que realizam o recebimento e tratamento de mensagens dos participantes do grupo são mostradas nas figuras 4.6 e 4.7.

Quando um participante recebe uma nova mensagem do protocolo de confirmação, envia uma resposta *confirm* ou *cancel* ao coordenador desta mensagem. Se este participante está enviando uma mensagem em 2PC, ou seja, também é um coordenador, ou está esperando a confirmação de uma mensagem 2PC recebida anteriormente, a resposta enviada é *cancel*. Caso contrário, este peer armazena a mensagem 2PC recebida, envia a resposta *confirm* e inicia uma contagem de tempo para o recebimento da confirmação desta mensagem.

A espera da confirmação é executada separadamente pela função *EsperaConfirmacao()*. Um *peer* participante estabelece um *timeout* para o recebimento da confirmação de uma mensagem 2PC. Após este tempo, o participante envia uma mensagem pedindo ao coordenador para reenviar a confirmação. O participante reinicia sua participação no grupo caso não receba a confirmação e o coordenador falhe. O participante reinicia para nao ficar bloqueado e prevenir inconsistência no caso de outros participantes terem recebido a confirmação do coordenador.

Quando a mensagem de confirmação é recebida, o *peer* participante age de acordo com a decisão contida nesta mensagem. Se a decisão recebida é *confirm*, o participante executa a ação contida na mensagem 2PC. Se a decisão é *cancel*, a mensagem 2PC é ignorada.

## Reenvio da Mensagem de Confirmação

Um *peer* pode receber uma mensagem requisitando o reenvio de confirmação de uma mensagem 2PC que tenha enviado previamente. Este *peer* pode já ter encerrado ou ainda estar executando a segunda fase do protocolo de confirmação desta mensagem. No caso deste *peer* já ter encerrado a execução do protocolo 2PC para esta mensagem, reenvia a

```
|| RecebeMensagens2PC(mensagem recebida: msg2PC):
/* Quando o peer i receber uma mensagem de um peer j */
/* O peer i pode ser tanto um coordenador
  quanto um participante para as mensagens recebidas */
  Caso msg seja:

    /* Mensagem recebida quando peer i age como coordenador */
    Mensagem de resposta do peer j para uma mensagem 2PC enviada pelo peer i
      /* Decisão feita pela função EsperaRespostas(), paralelamente */
      Guarda resposta do peer j para decisão

    /* Mensagem recebida quando peer i age como coordenador de alguma msg2PC */
    Mensagem de requisição de reenvio de confirmação
      Se este peer terminou o envio desta msg2PC
      /* Peer i ja encerrou o envio de confirmação da msg2PC */
        Obtem msg2PC e decisão da lista de transações enviadas
        Reenvia confirmação para peer j

    /* Mensagem recebida quando peer i age como participante */
    Nova mensagem msg2PC recebida do peer j:
      Se peer esta enviando ou esperando confirmação de mensagem 2PC
        Envia cancell para peerj
      Senão
        TrataNovaMsg2PC(msg2PC)

    /* Mensagem recebida quando peer i age como participante */
    Mensagem de confirmação para uma mensagem 2PC aguardando
      /* Para não causar inconsistencia: peer j não recebeu resposta do peer i */
      Se peer i não recebeu a mensagem 2PC para esta confirmação
        Reinicia este peer i
      Senão
        TrataConfirmacao(msg2PC)
```

Figura 4.6: Algoritmo de recebimento de mensagens do protocolo de confirmação.

```
|| TrataNovaMsg2PC(mensagem2PC: msg2PC):
/* Trata nova mensagem 2PC recebida */
  Envia confirm de msg2PC para peer j
  /* funcao EsperaConfirmacao(msg2PC) */
  Aguarda confirmação em paralelo


|| EsperaConfirmacao(mensagem2PC: msg2PC):
/* Peer i espera confirmação do peer j para mensagem 2PC recebida */
/* O peer j pode ter falhado: envia pedido de confirmação */
  Repete n vezes
    Espera chegar confirmação do peer j ou ocorrer timeout
      Se chegou confirmação
        Sai da repeticao
      Senao /* timeout */
        Envia mensagem ao peer j com pedido de confirmação de msg2PC
  Se não recebeu confirmação
    /* peer j pode ter falhado e enviado confirmação para alguns membros do grupo */
    /* Reinicia para não ficar bloqueado */
    Reinicia este peer i


|| TrataConfirmacao(mensagem2PC: msg2PC):
/* Trata confirmação da mensagem 2PC recebida anteriormente */
  Se (confirmacao = confirm)
    Entrega mensagem 2PC recebida para aplicacao
  Senao
    Ignora mensagem 2PC recebida
```

Figura 4.7: Funções auxiliares do algoritmo de recebimento de mensagens.

confirmação para o *peer* que requisitou. Caso contrário não reenvia, pois a mensagem de confirmação ainda será enviada por este *peer* a todos os membros do grupo.

## 4.4 Eleição de Líder

O módulo de eleição de líder é responsável por designar um *peer* do grupo para atender a requisições feitas por outros *peers*. Quando um *peer* faz uma requisição a um grupo, todos os membros deste do grupo a recebem, mas apenas um destes *peers* a atende. Todos os *peers* do grupo possuem as mesmas informações sobre cada requisição recebida, e determinam qual *peer* vai atendê-la. Este *peer* é chamado de servidor da requisição.

Um *peer* que deseja fazer uma requisição ao grupo deve utilizar a função *EnviaMsgRequisicao()*, mostrada na figura 4.8. Este *peer*, dito cliente, não precisa executar todas as funções do módulo de eleição, apenas as funções de envio de requisição. Esta função é disponibilizada de forma independente da comunicação em grupo, ou seja, o *peer* que a utiliza pode não fazer parte do grupo mantido pelo módulo da comunicação em grupo. Este *peer* age como cliente da requisição e será atendido por algum *peer* do grupo. O *peer* cliente deve enviar as informações necessárias para que tenha sua requisição atendida. Todos os *peers* do grupo mantêm uma lista com as requisições pendentes em um determinado instante de tempo.

O *peer* eleito atende a requisição recebida e ativa seu módulo local responsável por atender a requisição. No caso de falha deste *peer*, o grupo age de maneira a substituí-lo, procedimento que será descrito no decorrer desta seção. Após a requisição ser atendida, o mesmo *peer* cliente envia uma mensagem avisando que a requisição foi atendida. O envio desta mensagem é feito com o uso da função *EnviaMsgRequisicaoCompleta()*, mostrada na figura 4.8. Caso o cliente falhe antes do término da requisição, o próprio *peer* do grupo que está agindo como servidor envia mensagem ao grupo para que todos os *peers* removam a requisição das suas listas.

Os *peers* servidores devem ter todas as funções do módulo de eleição executando: recebimento de mensagens, eleição, supervisão e substituição do servidor em caso de falha. Estas funções são mostradas nas figuras 4.9, 4.11 e 4.10, e serão descritas na sequência.

```
/* Para enviar uma mensagem de Requisicao ao grupo */
/* Funcao chamada pela aplicacao do peer que faz a requisicao:cliente */
|| EnviaMsgRequisicao(mensagemAplicacao: msgAplicacao):
   Monta mensagemEleicao utilizando a msgAplicacao
   Envia mensagemEleicao ao grupo


/* Para enviar uma mensagem de Requisicao Completa ao grupo */
/* Funcao chamada pela aplicacao do peer que faz a requisicao:cliente */
|| EnviaMsgRequisicaoCompleta(mensagemAplicacao: msgAplicacao):
   Monta mensagemEleicao utilizando a msgAplicacao,
     com indicativo de requisicao completa
   Envia mensagemEleicao ao grupo
```

Figura 4.8: Envio de requisições ao grupo.

## Variáveis do Módulo de Eleição

As variáveis utilizadas pelas funções do módulo de eleição são mostradas abaixo: *esperandoMsgInicial*, *numeroRequisicoes*, *visao*, *lRequisicoesRecebidas* e *lRequisicoesAtendidas*. A variável *esperandoMsgInicial* indica que um *peer* enviou uma mensagem inicial quando começou a executar o módulo de eleição, mas ainda não recebeu resposta. Esta resposta é importante pois traz as requisições que ainda estão sendo atendidas pelo grupo.

O *numeroRequisicoes* é o número total de requisições já enviadas ao grupo e é utilizado para fazer o cálculo de qual *peer* deve atender uma requisição recebida. Sempre que uma nova requisição chega ao grupo, este valor é incrementado e associado a esta requisição. Todos os *peers* do grupo possuem o mesmo valor para o número de requisições, enviado nas mensagens do módulo de comunicação em grupo.

A *visão* do grupo é a lista de *peers* mantida pelo módulo de comunicação em grupo, utilizada para fazer o cálculo do *peer* servidor e também verificar se algum *peer* servidor falhou. Por fim, há duas listas que guardam as requisições em cada *peer*. A lista de requisições recebidas armazena a requisicao em si, o número que foi associado à requisição e a identificação do *peer* que foi designado para atendê-la. A lista de requisições atendidas armazena a requisição e é utilizada pela camada de aplicação.

```
/* Variaveis */
  boolean esperandoMsgInicial = true
  integer numeroRequisicoes = 0
  ListaDePeers visao = obtem visao do grupo da comunicacao em grupo
  ListaDeRequisicoes,NumeroDaRequisicao,PeerServidor lRequisicoesRecebidas = null
  ListaDeRequisicoes lRequisicoesAtendidas = null
```

Figura 4.9: Variáveis do algoritmo de eleição.

## Supervisão da Eleição

Os *peer* servidores iniciam a execução do módulo de eleição é através da função *SupervisaoEleicao()*, mostrada no pseudo-código da figura 4.10. A função de supervisão envia a mensagem inicial, obtém o número de requisições recebidas conforme informação do módulo de gestão em grupos e entra em um laço para verificar continuamente os *peers* servidores das requisições. A cada intervalo de tempo, o *peer i* verifica se as requisições da sua lista estão associadas a *peers j* falhos, ou seja, *peers* que não estão na visão local do grupo. Se algum *peer j* que está servindo uma requisição está falho, o *peer i* faz a troca deste *peer j* por outro, que esteja na visão do grupo.

A função de supervisão inicia um processo independente para tratar as mensagens recebidas pelo módulo de eleição. Após a inicialização, o *peer* está pronto para receber as mensagens e requisições de outros *peers*, executando a função *RecebeMsgEleicao()*, mostrada na figura 4.11 e descrita a seguir. Há quatro tipos de mensagens possíveis: mensagem com nova requisição, mensagem com aviso de requisição completa, mensagem de inicialização da eleição e mensagem de resposta para a inicialização.

## Inicialização

As mensagens de inicialização e de resposta da inicialização são enviadas por *peers* que fazem parte do grupo de servidores. A mensagem de inicialização do módulo de eleição é utilizada para pedir aos demais membros do grupo que enviem todas as requisições recebidas antes deste *peer* iniciar sua participação. Quando um *peer* recebe esta mensagem,

```
/* Faz a troca do peer servidor */
|| SupervisaoEleicao():
  esperandoMsgInicial = true
  numeroRequisicoes = obtem numero de requisicoes da comunicacao em grupo
  Inicia processo independente para funcao RecebeMsgsEleicao()
  Repete
    Se (ha apenas um membro no grupo)
      esperandoMsgInicial = false
    Senao
      Se (esperandoMsgInicial)
        envia mensagem inicial da eleicao
    Dorme por tempo t
    requisicoes = obtem lista de requisicoes recebidas
    Para cada requisicao[i] da lista requisicoes:
      Se (peer servidor da requisicao[i] nao esta em visao)
        TrocaServidor(requisicao[i])


/* Faz a troca do peer servidor para esta requisicao */
|| TrocaServidor(Requisicao: requisicao):
  nroNovoServidor = numeroDaRequisicao % numeroMembrosGrupo
  novoServidor = peer da posicao nroNovoServidor na visao do grupo
  Substitui o peer servidor por novoServidor em requisicao
```

Figura 4.10: Funções de supervisao da eleição.

envia a resposta ao iniciante, com uma lista de todas as requisições recebidas, o número de cada requisição e qual *peer* a está atendendo. As requisições enviadas são as que estão na lista de requisições recebidas do *peer*. Quando o *peer* inicializando recebe a resposta da mensagem inicial, armazena as requisições recebidas na sua lista local *lRequisicoesRecebidas* e altera seu estado (*esperandoMsgInicial*) para indicar que já recebeu esta resposta de algum *peer* do grupo.

## Escolha do Servidor

As mensagens de nova requisição e de aviso de requisição completa são enviadas aos *peers* do grupo por *peers* cliente, como dito anteriormente. Quando um *peer* servidor recebe mensagem com nova requisição, incrementa o número de requisições recebidas, atualiza este número no módulo de comunicação em grupo e faz a eleição de qual *peer* deve atender esta requisição.

A eleição, ou a atribuição de qual *peer* será o servidor é feita com base no número de requisições já enviadas ao grupo, e na quantidade de *peers* do grupo, como mostrado na figura 4.11. O resto da divisão inteira (módulo) do número de requisições pela quantidade de *peers* do grupo é utilizado para definir o *peer* servidor da requisição em questão. A expressão matemática que realiza este cálculo é: *indiceServidor = numeroDaRequisicao MOD numeroMembrosGrupo*. Chamando o resto da divisão de índice do servidor, o *peer* da posição do índice do servidor na visão, irá atender a requisição. Todos os membros do grupo mantém uma lista ordenada com os *peers*, a visão, mantida adequadamente pelo módulo de comunicação em grupo.

Além da definição do servidor, a função *Eleicao()* insere a requisição, o número da requisição e a identificação do *peer* servidor em uma lista local de requisições recebidas *lRequisicoesRecebidas*. Esta lista é utilizada para fazer a supervisão da eleição, como dito anteriormente. Ainda na função *Eleicao()*, se o *peer* eleito é o próprio *peer* em questão, a requisição é inserida na lista de requisições atendidas *lRequisicoesAtendidas*. Esta lista é entregue para a camada de aplicação, que cuidará para que a requisição seja atendida adequadamente.

```
|| RecebeMsgEleicao(MensagemEleicao: msgEleicao):
/* Quando o peer i receber uma requisição de um peer j */
/* Mensagens recebidas pelos peers do grupo: servidores */
  Caso msgEleicao seja:

    Mensagem de requisição
    /* Nova requisição por um download: recebida por todo o grupo */
      numeroRequisicoes = numeroRequisicoes +1
      Atualiza o numero de requisicoes no modulo comunicação em grupo
      Eleicao(msgEleicao)

    Mensagem de requisição completa
    /* Aviso de download completo: recebido por todo o grupo */
      Remove requisição da lista lRequisicoesRecebidas */

    Mensagem inicial
    /* Mensagem inicial do modulo de eleção: recebida por todos o grupo */
      Envia mensagem para peer j com todas as requisicoes da
        lista lRequisicoesRecebidas

    Mensagem resposta da mensagem inicial
    /* Contem lista de requisicoes do grupo:
      mensagem recebida pelo peer que esta iniciando */
      esperandoMsgInicial = false
      Adiciona as requisicoes recebidas na mensagem na lista
        local lRequisicoesRecebidas
      Se peer i e' servidor de alguma requisição recebida
        Insere requisição da msgEleicao em lRequisicoesAtendidas
        Avisa aplicação para atender requisição


|| Eleicao(MensagemEleicao: msgEleicao):
/* Atribuição de um peer servidor para a nova requisição que chegou */
  /* numeroRequisicoes foi incrementado: numero da nova requisição */
  nroPeerServidor = numeroRequisicoes % numeroMembrosGrupo
  meuNumero = posição do peer i na visão do grupo
  Se (nroPeerServidor = meuNumero)
    peerServidor = identificação do peer i
    Insere requisição da msgEleicao em lRequisicoesAtendidas
    Avisa aplicação para atender requisição
  Senão
    peerServidor = peer da posição nroPeerServidor na visão do grupo
  Insere requisição da msgEleicao, numeroRequisicoes, peerServidor em
    lRequisicoesRecebidas
```

Figura 4.11: Recebimento de mensagens e eleição do servidor.

No caso de falha de um *peer* servidor, a substituição é realizada pela função de supervisão da eleição, dos outros *peers* servidores. Quando a falha de *peer* servidor é detectada por outro *peer* servidor, o processo de supervisão faz a troca do *peer* servidor por um *peer* que está na visão do grupo do *peer* que está executando. A eleição do novo servidor é realizada como no procedimento de eleição. Ou seja, o índice do servidor é calculado com base no número da requisição e na quantidade de *peers* no grupo, e o *peer* eleito é aquele que está na posição deste índice na visão do grupo. O *peer* executando substitui o servidor falho pelo novo *peer* servidor, na sua lista de requisições recebidas. Caso o próprio *peer* seja o novo servidor, inicia o tratamento da requisição.

## Requisição Completa

Outro tipo de mensagem que pode ser recebida é de aviso de requisição completa. Quando um *peer* do grupo recebe esta mensagem, remove a requisição em questão da sua lista de requisições recebidas. Não há necessidade do *peer* servidor remover a requisição da lista de requisicoes atendidas, pois esta lista é utilizada pela aplicação da forma que for necessário.

# Capítulo 5

# Implementação JXTA e Estudo de Caso

Este capítulo apresenta um estudo de caso do uso da ferramenta desenvolvida neste trabalho. Uma aplicação P2P confiável para compartilhamento de arquivos foi construída, utilizando a plataforma JXTA e a ferramenta desenvolvida.

A seção 5.1 apresenta a aplicação de compartilhamento de arquivos, descrevendo sua estrutura, funcionalidade e implementação JXTA. A seção 5.2 descreve os experimentos realizados, mostrando os resultados obtidos para latência de todos os módulos da ferramenta apresentada no capítulo anterior.

## 5.1 Implementação JXTA

Tanto a ferramenta quanto a aplicação exemplo foram implementadas na linguagem Java [33], utilizando a plataforma JXTA. Foi utilizada a versão 1.5 da linguagem Java, e a versão 2.4.1 do JXTA para a criação da estrutura da rede P2P acima da qual a ferramenta executa.

O objetivo da aplicação de compartilhamento de arquivos desenvolvida é formar grupos de *peers* que disponibilizam arquivos de maneira confiável. Um *peer* que inicia o *download* do arquivo a partir de um grupo, tem a confiança de que irá receber todo o arquivo, independentemente da falha do membro do grupo que está lhe enviando o arquivo.

A figura 5.1 mostra a localização da ferramenta e da aplicação, dentro da estrutura do JXTA. A ferramenta e a aplicação P2P para compartilhamento de arquivos se localizam na camada de aplicações do JXTA, e utilizam os serviços que o núcleo desta plataforma oferece. Estes serviços incluem a criação dos *peers*, os quais possuem identificadores únicos, criação de canais de comunicação entre *peers* (*pipes*), a formação da estrutura de grupos de *peers* JXTA e a publicação e busca de recursos na rede JXTA.

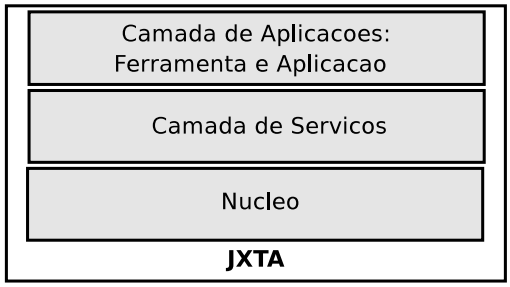


Figura 5.1: Localização da ferramenta dentro da estrutura do JXTA.

O próprio JXTA possui o conceito de grupos de *peers* que, entretanto, são diferentes dos grupos confiáveis formados pela aplicação proposta. Neste capítulo o termo *grupo JXTA* é utilizado para grupos de *peers* disponibilizados pela plataforma, e os termos *grupo da aplicação* e *grupos de arquivos* são utilizados para grupos formados pela ferramenta e aplicação propostas.

Os *peers* que executam os protocolos JXTA se organizam em grupos JXTA. Quando um *peer* inicia, automaticamente começa a fazer parte do grupo principal da estrutura do JXTA. A partir deste grupo JXTA, o *peer* pode criar outros grupos ou iniciar sua participação em grupos JXTA já existentes. A estrutura de grupos da aplicação será descrita nas próximas seções.

Os *peers* podem se comunicar através de canais de comunicação chamados *pipes*. Para que a comunicação seja possível, um *peer* deve possuir informações de identificação sobre o *pipe* do *peer* com o qual deseja se comunicar. Estas informações são fornecidas por *advertisements*. Os *advertisements* contém os dados necessários para a utilização dos recursos da rede JXTA. Cada recurso da rede P2P JXTA é descrito por um *advertisement*.

Para que a utilização de recursos de um *peer* por outro seja possível, é necessário que o proprietário deste recurso crie um *advertisement* e o publique na rede JXTA. Depois da publicação, os *peers* interessados podem fazer uma busca e utilizar o recurso a partir das informações contidas no *advertisement* encontrado.

A publicação e pesquisa no JXTA conta com o auxílio de *superpeers*, chamados de *rendezvous*. Os *rendezvous* armazenam índices que facilitam a localização dos recursos na rede. Quando um *peer* publica um *advertisement*, o rendezvous armazena o identificador do *peer* que possui o recurso correspondente àquele *advertisement*.

Todo *peer* deve se manter conectado a pelo menos um *rendezvous*. Para iniciar uma busca, o *peer* envia uma requisição ao *rendezvous* ao qual está conectado e aguarda respostas. Os *rendezvous* verificam seus índices e, caso necessário, propagam a busca para os *rendezvous* vizinhos. Se o recurso for encontrado, o *peer* que o possui responde a requisição.

As buscas por recursos são realizadas no contexto dos grupos JXTA. Isto significa que um *peer* pode fazer buscas dentro dos grupos JXTA dos quais participa, e apenas recursos publicados dentro do contexto destes grupos JXTA serão encontrados.

### 5.1.1 Hierarquia de Grupos JXTA da Aplicação Exemplo

O grupo principal da estrutura do JXTA é chamado de *World Peer Group* ou *Net Peer Group*, como mostrado na figura 5.2. Outros grupos podem ser criados a partir deste. Quando um *peer* inicia a execução dos protocolos do JXTA, automaticamente começa a fazer parte do grupo principal. A partir deste momento, os *peers* que desejam utilizar a aplicação de compartilhamento de arquivos fazem uma busca pelo *grupo da aplicação* e se conectam. Finalmente, dentro do grupo da aplicação estão localizados os *grupos de arquivos*, exemplificados na figura pelos grupos *A, B e C*.

Os grupos de arquivos são formados pelos *peers* que possuem cópias de um arquivo e desejam compartilhá-lo. Estes grupos são os responsáveis por manter e disponibilizar arquivos aos demais *peers* da rede. Cada grupo de arquivo disponibiliza um único arquivo. Os *peers*, por sua vez, podem fazer parte de quantos grupos de arquivo desejarem, de

acordo com a quantidade de arquivos quiserem disponibilizar.

O grupo da aplicação e os grupos de arquivos são grupos JXTA, dentro dos quais todos os membros executam a aplicação de compartilhamento de arquivos, como anteriormente mencionado. Os grupos JXTA são formações padrão da estrutura da plataforma JXTA, e fornecem a estrutura para a execução da ferramenta desenvolvida. O controle efetivo de membros e a comunicação em grupos é realizada pela ferramenta desenvolvida, dentro de cada grupo de arquivo.

Na inicialização da aplicação, cada *peer* faz uma busca pelo grupo da aplicação, dentro do grupo principal do JXTA. Caso não seja encontrado, o grupo é criado. Isto significa que não há mais *peers* acessíveis executando a aplicação. Caso seja encontrado, o *peer* se conecta ao grupo da aplicação.

Os próximos passos de execução dependem se o *peer* deseja compartilhar arquivos ou fazer *download* de arquivos de outros *peers*, conforme detalhado nas próximas seções. Se o *peer* deseja obter arquivos, faz a busca pelo grupo do arquivo correspondente. Se um *peer* deseja disponibilizar um arquivo, faz a busca pelo grupo do arquivo correspondente e se conecta, ou cria um novo grupo caso ainda não exista.

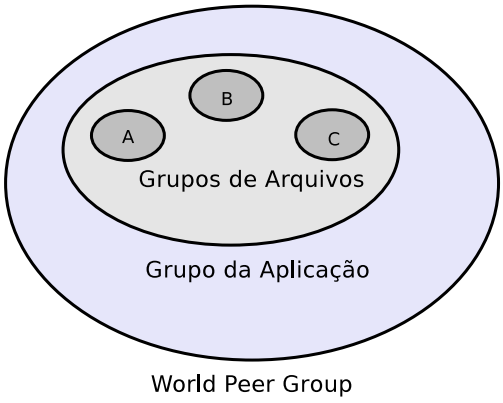


Figura 5.2: Estrutura de grupos da aplicação no JXTA.

### 5.1.2 *Peer* Servidor

Os *peers* que disponibilizam arquivos a outros *peers* da rede são chamados de servidores. Após se conectarem ao grupo da aplicação, os *peers* servidores se conectam ao grupo do arquivo correspondente ao arquivo que deseja disponibilizar. Se este grupo ainda não existir, o *peer* o cria e o publica. Os *peers* servidores podem permanecer nos grupos de arquivo o tempo que desejarem.

Os *peers* membros de grupos de arquivo executam a aplicação de compartilhamento de arquivos. A aplicação de compartilhamento de arquivos, por sua vez, utiliza a ferramenta para as tarefas de manutenção de membros do grupo e de eleição de um *peer* para atender cada requisição de *download* recebida. Para cada grupo de arquivos, a comunicação em grupos faz o controle de membros separadamente. Os *peers* terão uma instância da ferramenta executando, para cada grupo de arquivo do qual fizer parte. Em outras palavras, para cada grupo de arquivo do qual for membro, o *peer* executa os módulos de comunicação em grupo e eleição de líder da ferramenta. A aplicação de compartilhamento de arquivos, como um todo, pode possuir várias instâncias da ferramenta executando, para que um *peer* possa fazer parte de mais de um grupo de arquivo.

Os grupos de arquivos fornecem uma interface com a qual outros *peers* da rede podem obter o arquivo mantido pelo grupo. Os *peers* que desejam obter arquivos, chamados de clientes, utilizam esta interface para enviar uma requisição de *download* ao grupo do arquivo. Quando um grupo recebe uma requisição de *download*, elege um *peer* para enviar este arquivo ao cliente. A eleição é realizada pelo módulo de eleição de líder executado pelos *peers* do grupo.

Após a eleição, o *peer* designado para atender a requisição, chamado de servidor da requisição, inicia o envio do arquivo ao cliente. Caso o servidor de uma requisição falhe ou saia do grupo durante o envio, os demais membros do grupo deste arquivo elegem outro servidor. Após o final do envio, o grupo do arquivo recebe uma mensagem de confirmação de requisição completa enviada pelo cliente.

### 5.1.3 *Peer* Cliente

Os *peers* cliente também se conectam ao grupo da aplicação para procurar por grupos de arquivos disponíveis. Estes *peers* fazem parte apenas do grupo da aplicação, e utilizam a funcionalidade de *download* que os grupos de arquivos oferecem.

Para cada arquivo que desejar, o cliente deve fazer uma busca. Caso encontre o grupo com o arquivo desejado, este *peer* envia uma requisição de *download*, utilizando a função de envio de requisição. O cliente não precisa executar os demais módulos da ferramenta, apenas a função de envio de requisição.

A requisição enviada ao grupo contém a informação (*advertisement*) do canal de comunicação (*pipe*) que o servidor deve utilizar para enviar o arquivo ao cliente. Após enviar a requisição, o cliente cria um *pipe* de escuta e espera pelo recebimento de mensagens do servidor. Depois de receber todo o arquivo, o cliente encerra este *pipe* e envia uma mensagem de confirmação de requisição completa ao grupo, para informar que o *download* foi realizado.

## 5.2 Estudo de Caso

Os resultados descritos nesta seção foram obtidos através de um estudo de caso detalhado a seguir. Foram utilizados cinco computadores, cada um executando várias instâncias de *peers*. Oito *peers* servidores foram executados em quatro máquinas, ou seja, cada computador executou duas instâncias de *peers* servidores. O quinto computador foi utilizado para a execução de um *peer rendezvous* e dos *peers* cliente. Os *peers* cliente foram utilizados durante os testes do módulo de eleição de líder.

Para as simulações de *download* deste estudo de caso, foi utilizado sempre o mesmo arquivo, fazendo com que os oito *peers* servidores pertencessem ao mesmo grupo de arquivo. O clientes fizeram *downloads* deste arquivo. O módulo de comunicação em grupos da ferramenta foi parametrizada para utilizar um intervalo de tempo de 10 segundos para envio de mensagens com a visão. Os experimentos foram repetidos diversas vezes e os resultados apresentados são representativos do conjunto de resultados obtidos.

Para que seja possível a execução de mais de um *peer* por computador, cada *peer* utiliza uma porta de comunicação diferente. O JXTA faz este controle, sendo necessário apenas executar cada *peer* a partir de diretórios distintos do computador.

Foi inserido um atraso no envio de cada mensagem, pois os computadores estavam conectados através de uma rede local e desejava-se obter um cenário com *peers* executando em redes heterogêneas. Este atraso varia de zero a cem milisegundos, distribuído de maneira uniforme neste intervalo.

### 5.2.1 Latência do Módulo de Comunicação em Grupos

Esta seção mostra resultados obtidos para a latência na detecção da entrada e saída de membros do grupo. Para obter os resultados de detecção de entrada de membros no grupo, os 8 *peers* foram iniciados um a um. O procedimento padrão adotado foi inicializar um *peer* apenas quando os *peers* que entraram anteriormente já haviam sido detectados pelos membros do grupo.

Para medir a latência de entrada, foi computado o tempo decorrido entre o envio da mensagem inicial do módulo de comunicação em grupos pelo *peer* que iniciou, até a inclusão deste *peer* na visão local de cada membro. Estas diferenças de tempo foram somadas e divididas pela quantidade de membros que detectaram a entrada do *peer* no grupo. Foram considerados também os valores obtidos na execução dos testes dos módulos do protocolo de confirmação e eleição. Foi iniciado um *peer* em cada computador para depois iniciar o segundo *peer* em uma máquina. O resultado é mostrado na figura 5.3, considerando a quantidade de peers no sistema, no momento de cada detecção.

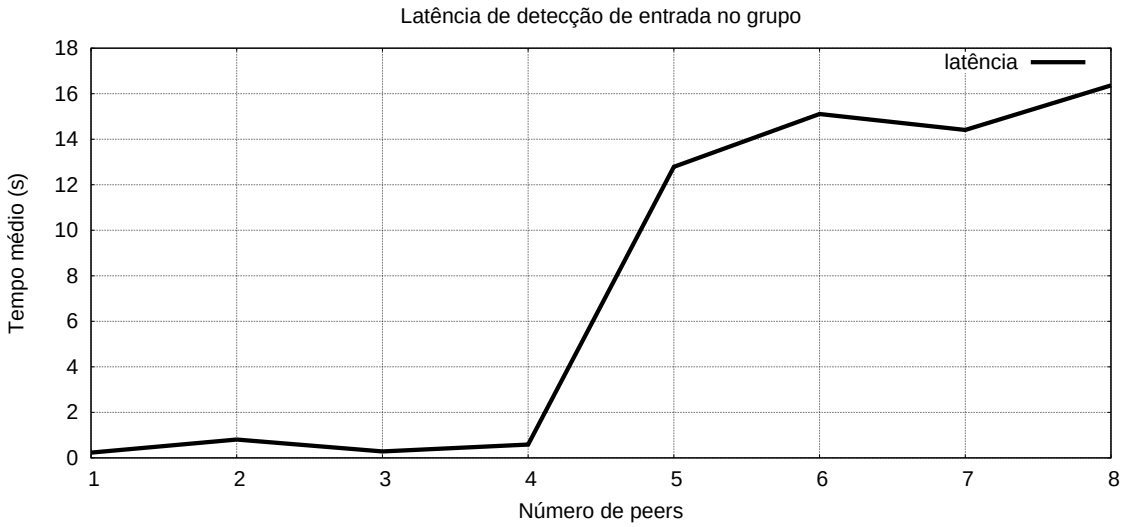


Figura 5.3: Latência na detecção de entrada de membros no grupo.

A figura 5.3 mostra a latência média de detecção na entrada um membro em um grupo, de acordo com a quantidade de *peers* do grupo. Este valor foi variável para este estudo de caso, dependendo da carga de processamento em cada computador. Nota-se um aumento considerável da latência, quando o segundo *peer* é iniciado em um computador, ou seja, a partir de quatro membros no grupo. No momento da entrada de um novo *peer* no grupo, os membros continuam executando todas as funcionalidades da aplicação. Estas funcionalidades são executadas por processos separados, que competem para a utilização do mesmo processador.

A média da latência de detecção de novos membros obtida para este estudo de caso foi de 11.8 segundos. A comunicação em grupos é baseada em intervalos de tempo para a execução de tarefas. Nesta implementação, o intervalo de tempo utilizado foi de 10 segundos. Em termos destes intervalos, a latência na detecção de novos membros é de 1.2 intervalos de tempo. Logo, estes valored de latência estão dentro do esperado.

Os resultados de latência para a detecção de saída de um membro do grupo foram obtidos através da retirada de cada *peer* do grupo, um a um. Cada membro foi retirado do grupo após alguns *peers* já terem detectado a falha do membro retirado anteriormente. O encerramento da execução de um *peer* foi feito após este *peer* e a maioria dos demais

membros do grupo já terem detectado a falha do *peer* que saiu anteriormente. Os valores obtidos na execução dos testes dos módulos do protocolo de confirmação e eleição também foram considerados.

O gráfico apresentado na figura 5.4 mostra os resultados médios para latência na detecção da falha de membros do grupo, de acordo com a quantidade de *peers*. De maneira geral, a latência na detecção da saída de membros do grupo aumentou suavemente acompanhando o aumento de *peers* no grupo. A concentração dos valores para a latência ficou entre 11 e 13.5 segundos, com média de 12.8 segundos. Comparado com intervalos de tempo, este valor significa 1.3 intervalos de tempo para a detecção da falha de um membro, por todos os demais *peers* do grupo.

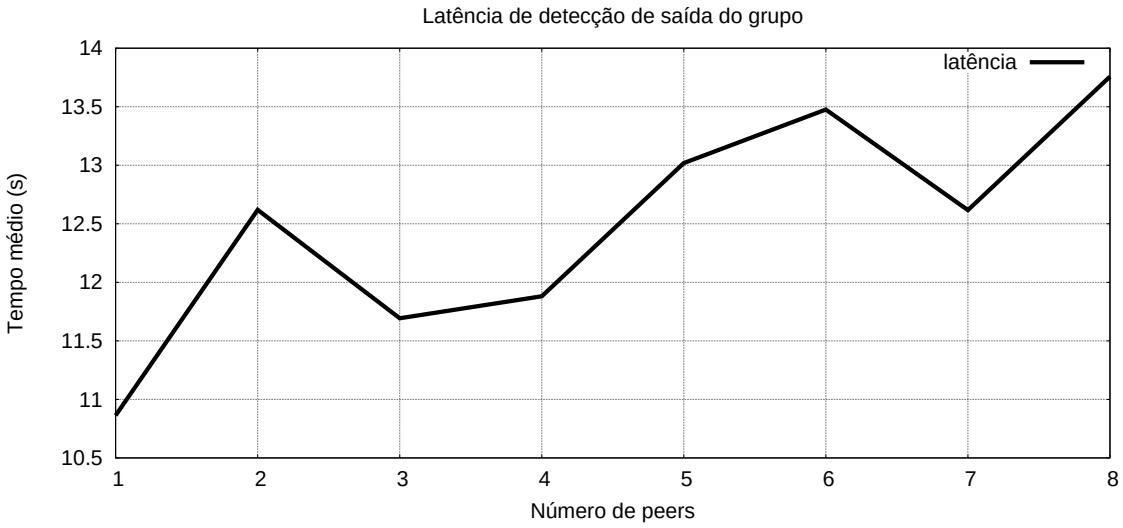


Figura 5.4: Latência na detecção da saída de membros do grupo.

### 5.2.2 Latência do Módulo do Protocolo de Confirmação

O módulo que executa o protocolo de confirmação para mensagens enviadas por membros do grupo foi avaliado quanto ao tempo decorrido entre o envio da mensagem pelo *peer* coordenador e o recebimento da confirmação desta mensagem, pelos membros do grupo. Mensagens 2PC foram enviadas variando a quantidade de membros do grupo.

O gráfico mostrado na figura 5.5 apresenta os resultados de latência obtidos a partir

deste estudo de caso. A média da latência, independente da quantidade de *peers* foi de 6.4 segundos. Considerando a quantidade de *peers* no grupo no momento do envio da mensagem e recepção da confirmação, a latência variou entre 5 e 8 segundos.

Após enviar uma mensagem pelo protocolo de confirmação, o coordenador da mensagem aguarda um certo tempo até receber a resposta de todos os membros do grupo. O tempo utilizado neste estudo de caso foi a metade do intervalo de tempo do módulo de comunicação em grupos, ou seja, 5 segundos. Este tempo foi suficiente nos casos onde a visão do grupo não se alterou durante a execução do protocolo. No entanto, o valor médio para a latêcia aumentou, pois a ocorrência de falhas exige que o 2PC aguarde a obtenção da visão atualizada pelo módulo do protocolo de confirmação.

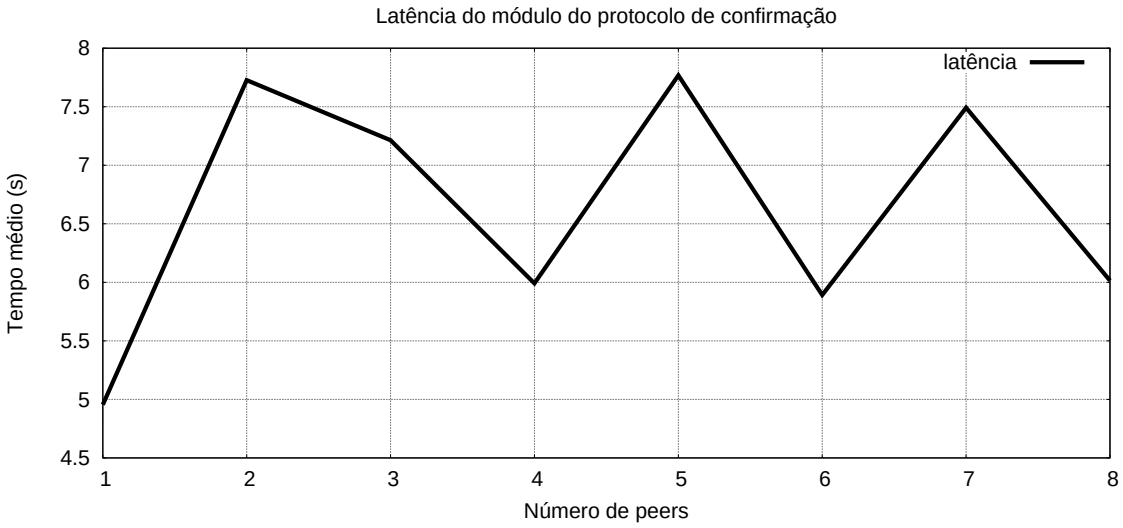


Figura 5.5: Latência na entrega da mensagem ao grupo.

### 5.2.3 Latência do Módulo de Eleição de Líder

O módulo de eleição de líder foi medido quanto à latência entre o envio de uma requisição por um *peer* e a efetiva eleição de um *peer* do grupo, para atender esta requisição. Casos de falha do *peer* que está servindo uma requisição também foram analisados. Enquanto atendiam uma requisição, *peers* foram retirados do grupo, para que o tempo necessário para a substituição de um *peer* servidor também fosse computado. A quantidade de *peers*

no grupo variou durante os experimentos.

O gráfico apresentado na figura 5.6 mostra a latência obtida para este estudo de caso. A eleição de membros do grupo acontece de forma descentralizada nos membros do grupo, o que faz com que o tratamento das requisições aconteça assim que os *peers* recebem a requisição. O aumento da latência ocorre quando há a falha do *peer* servidor, pois a substituição ocorre apenas após a detecção desta falha.

Neste estudo de caso, a latência variou entre 7 e 13 segundos, com média em 11 segundos. O valor da média significa pouco mais de um intervalo de tempo do módulo de comunicação em grupos utilizado neste estudo de caso. De fato, o valor do intervalo de tempo influencia no tempo necessário para a detecção das falhas de membros do grupo, que por sua vez, influencia na latência da substituição de um *peer* servidor falho.

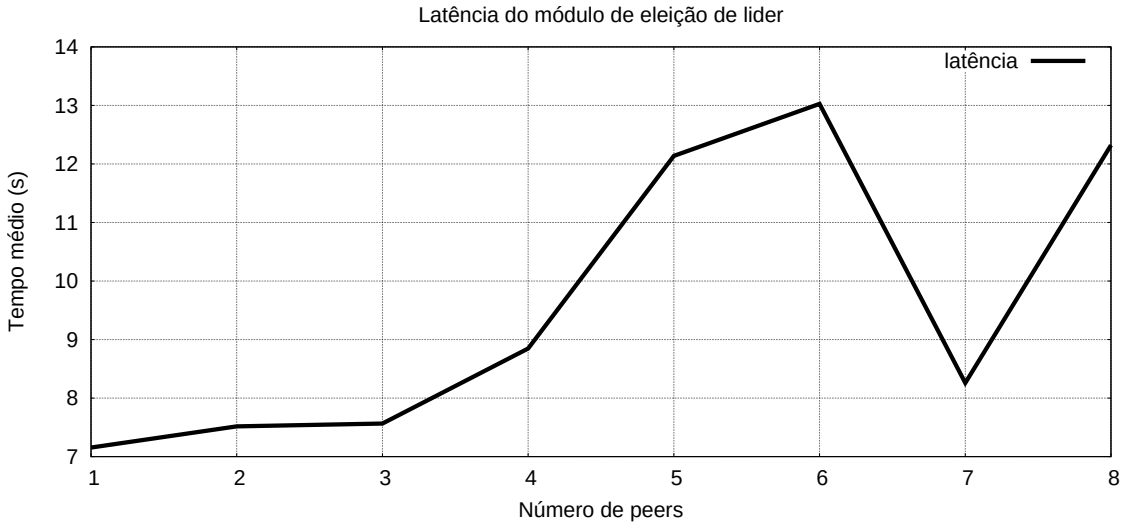


Figura 5.6: Latência na eleição do peer para atender uma requisição.

# Capítulo 6

# Conclusão

Este trabalho apresentou uma ferramenta para a construção de sistemas P2P confiáveis, que fornece um serviço de comunicação em grupos com protocolo de confirmação e eleição, implementado na plataforma JXTA. Uma aplicação para compartilhamento de arquivos foi construída, com base na ferramenta desenvolvida. Inicialmente foram descritas as principais características dos sistemas P2P e suas possíveis organizações e propriedades, além de tipos de aplicação existentes e alguns exemplos. A seguir foi apresentada a biblioteca para desenvolvimento de sistemas P2P JXTA, com sua arquitetura e funcionalidades. Este trabalho também examinou as propriedades dos sistemas distribuídos tolerantes a falhas, apresentou protocolos de acordo e confirmação, técnicas de replicação e o serviço de comunicação em grupos. Algumas ferramentas que fornecem o serviço de comunicação em grupos foram descritas brevemente.

A ferramenta desenvolvida por este trabalho implementou um serviço de comunicação em grupos que pode ser utilizado para a construção de sistemas P2P. O grupo de *peers* disponibiliza recursos aos usuários de forma transparente. Com a presença de membros sem-falha, a falha de alguns membros do grupo não causa interrupção do fornecimento dos serviços. A implementação de um algoritmo de eleição de líder permite determinar qual componente do grupo atende a cada requisição, enquanto o protocolo de confirmação em duas fases permite a entrega atômica de mensagens. A aplicação P2P para compartilhamento de arquivos, implementada a partir da ferramenta, também foi apresentada.

Foram descritos a sua estrutura, a implementação JXTA e a interação entre *peers* para fazer o *download* de arquivos. Por fim, foi apresentado um estudo de caso para avaliação da ferramenta e da aplicação construídas, mostrando resultados de latência para todos os módulos da ferramenta.

Uma avaliação do impacto da ferramenta no desempenho da rede deve ser realizada em trabalhos futuros. A implementação de protocolos de ordenação da entrega de mensagens pode ser feita futuramente, para aumentar a funcionalidade da ferramenta e contemplar a construção de diferentes sistemas, tais como edição compartilhada de arquivos. A plataforma JXTA, sobre a qual este trabalho foi desenvolvido, fornece serviços que não foram utilizados nesta implementação, como o protocolo de informações (PIP). Um estudo deste protocolo pode ser realizado em trabalhos futuros, para avaliar seu emprego como uma forma alternativa de obtenção de dados sobre os *peers* da rede.

# Referências Bibliográficas


[1] Anderson, D.P. et al., "SETI@home: An Experiment in Public-Resource Computing," *Communications of the ACM*, vol. 45 n. 11, pp. 56-61, 2002.

[2] Ban, B., "Design and implementation of a reliable group communication toolkit for java," *Cornel University*, 1998.

[3] Banerjee, D., Saha, S., Sen, S., Dasgupta, P., "Reciprocal Resource Sharing in P2P Environments," *Proceedings of The Fourth International Joint Conference on Autonomous and Agents & Multi Agents Systems (AAMAS'05)*, 2005.

[4] Bauer, M.A., Wang, T., "Strategies for Distributed Search," *Proceedings of The 1992 ACM Annual Conference on Communications*, pp. 251-260, 1992.

[5] Bernstein, P.A., Hadzilacos, V., Goodman, N., "Concurrency Control and Recovery in Database Systems," Addison-Wesley, 1987.

[6] Birman, K.P., Van Renesse, R., "Reliable Distributed Computing with the Isis Toolkit," *IEEE Computer Society Press*, 1994.

[7] "BitTorrent," *http://www.bittorrent.com/*, acesso em julho de 2006.

[8] Brookshier, D., Govoni, D., Krishnan, N., Soto, J.C., "JXTA: Java P2P Programming," *Sams Publishing*, 2002.

[9] Budhiraja, N., Marzullo, K., Schneider, F., Toueg,S., "Distributed Systems," ch. The Primary-Backup Approach, pp. 199-216, *Addison-Wesley*, 1993.

[10] Castro, M., Druschel, P., Kermarrec, A., Rowstron, A., "SCRIBE: A large-scale and decentralized application-level multicast infrastructure," *IEEE Journal On Selected Areas In Communications*, vol. 20, n. 8, pp. 101-103, 2001.

[11] Chandra, T., Toueg, S., "Unreliable failure detectors for reliable distributed systems," *Journal of ACM,* vol. 43, pp. 225-267, 1996.

[12] Chawathe, Y., Ratnasamy, S., Breslau, L., Lanham, N., Shenker, S., "Making Gnutella-like P2P Systems Scalable," *Proceedings of The ACM SIGCOMM 2003*, 2003.

[13] Chockler, G.V., Keidar, I., Vitenberg, R., "Group Communication Specifications: A Comprehensive Study," *ACM Computing Surveys*, vol. 33, pp.427-469, 2001.

[14] Comer, D.E., Stevens, D.L., "Internetworking With TCP/IP Volume III: Client-Server Programming and Applications," *Prentice Hall*, 2000.

[15] "Communication and Collaboration Systems," *http://research.microsoft.com/ccs/*, acesso em julho de 2006.

[16] Défago, X., Schiper, A., Sergent, N., "Semi-passive replication," *in Proceedings of The 17th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pp. 43-50, 1998.

[17] Défago, X., Schiper, A., Urbán, P., "Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey," *ACM Computing Surveys*, vol. 4, pp.372-421, 2004.

[18] Doval D., O'Mahony, D. "Overlay Networks - A Scalable Alternative for P2P," *IEEE*, 2003.

[19] Eastlake, D., Reagle, J., Solo, D. "(Extensible Markup Language) Xml-Signature Syntax and Processing," *RFC Editor*, 2002.

[20] "eDonkey", *http://www.edonkey2000.com/*, acesso em julho de 2006.

[21] "eMule," *http://www.emule-project.net/*, acesso em julho de 2006.

[22] Fischer, M.J., Lynch, N.A., Paterson, M.S., "Impossibility of Distributed Consensus with one Faulty Process," *Journal of ACM*, vol. 32, pp. 374-382, 1985.

[23] Foster, I., "What is the Grid. A Three Point Checklist," *GridToday*, 2002.

[24] Foster, I., Iamnitchi, A., "On death, taxes, and the convergence of peer-to-peer and grid computing," *In Proceedings of 2nd International Workshop on Peer-to-Peer Systems (IPTS'02)*, 2003.

[25] "Gnutella", *http://www.gnutella.com/*, acesso em julho de 2006.

[26] Gradecki, J.D., "Mastering JXTA: Building Java Peer-to-Peer Applications," *John Wiley & Sons Publications*, 2002.

[27] Greve, F., Narzul, J.P.L., "Um protocolo de validação atômica não-bloqueante eficiente," *SBRC 2002*, pp. 309-323, 2002.

[28] Greve, F.G.P., "Protocolos Fundamentais para o Desenvolvimento de Aplicações Robustas," *SBRC'05*, 2005.

[29] Gray, J.N., "Notes on Database Operating Systems," *Lecture Notes in Computer Science*, vol. 60, pp. 393-481. Springer-Verlag, 1978.

[30] Hiltunen, M.A., "Configurable Fault-Tolerant Distributed Services," Ph.D. Dissertation, *The University of Arizona*, 1996.

[31] Jalote, P., "Fault Tolerance in Distributed Systems," *Prentice Hall*, 1994.

[32] Jansch-Pôrto, I.E.S., "Fundamentos de Tolerância a Falhas," *Revista de Informática Teórica e Aplicada,* vol. 1, n. 3, pp. 63-99, 1991.

[33] "Java Technology, " *http://www.sun.com/java/*, acesso em fevereiro de 2007.

[34] "JXTA-RM: Reliable Multicast in JXTA," *http://jxta-rm.jxta.org/*, acesso em julho de 2006.

[35] "JXTA Project," *http://www.jxta.org/*, acesso em julho de 2006.

[36] "Jxta v2.3x: Java Programmer's Guide," *http://www.jxta.org/docs/JxtaProgGuide_v2.3.pdf*, acesso em julho de 2006.

[37] "JXTA v2.0 Protocols Specification," *http://spec.jxta.org/nonav/v1.0/docbook/JXTAProtocols.html*, acesso em julho de 2006.

[38] "Kazaa," *http://www.kazaa.com*, acesso em julho de 2006.

[39] Lamport, L., Shostak, R., Pease, M., "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, pp. 382-401, 1982.

[40] Larrea, M., Fernández, A., Arévalo, S., "On The Implementation of Unreliable Failure Detectors in Partially Synchronous Systems," *IEEE Transactions on Systems*, 2004.

[41] Lui, S.M., Kwok, S.H., "Interoperability of Peer-to-Peer File Sharing Protocols," *ACM SIGecom Exchanges*, vol. 3, n.3, pp. 25-33, 2002.

[42] Lv, Q., Cao, P., Cohen, E., Li, K., Shenker, S., "Search and replication in unstructured peer-to-peer networks," *In Proceedings of the 16th ACM International Conference on Supercomputing (ICS'02)*, 2002.

[43] Lynch, N.A., "Distributed Algorithms," *Morgan Kaufmann*, 1996.

[44] Milojicic, D.S. et al, "Peer-to-Peer Computing," *HP Laboratories Palo Alto*, 2002.

[45] "Napster," *http://www.napster.com/*, acesso em julho de 2006.

[46] Nejdl, W. et al., "Edutella: A p2p networking infrastructure based on rdf," *In Proceedings of the 12th International Conference on Word Wide Web*, 2003.

[47] Pang, J. et al., "Availability, Usage, and Deployment Characteristics of the Domain Name System," *IMC'04*, 2004.

[48] Powell, D., "Delta-4: a Generic Architecture for Dependable Distributed Computing," *Springer-Verlag*, 1991.

[49] Renesse, R.V., Birman, K., Maffeis, S., "Horus: a flexible group communication system," *Communications of the ACM*, vol. 39, pp. 76-83, 1996.

[50] Rowstron, A., Druschel, P., "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems," *In Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, 2001.

[51] Samant, K., Bhattacharyya, S., "Topology, Search, and Fault Tolerance in Unstructured P2P Networks," *In Proceedings of the 37th Hawaii International Conference on System Sciences*, 2004.

[52] Schneider, F., "Distributed Systems," ch. Replication Management using the State Machine Approach, pp. 169-198, *Addison-Wesley*, 1993.

[53] Schollmeier, R., "A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications," *In Proceedings of First IEEE International Conference on Peer-to-Peer Computing (P2P'01)*, pp. 101-103, 2001.

[54] Srisuresh, P., Egevang, K., "Traditional IP Network Address Translator (Traditional Nat)," RFC Editor, 2001

[55] "Sun Microsystems," *http://www.sun.com/software/jxta/index.xml*, acesso em julho de 2006.

[56] "The Spread Toolkit," *http://www.spread.org*, acesso em julho de 2006.

[57] Theotokis, S.A., Spinellis, D., "A Survey of Peer-to-Peer Content Distribution Technologies," *ACM Computing Surveys*, vol. 36, n. 4, pp. 335-371, 2004.

[58] Turek, J., Shasha, D., "The Many Faces of Consensus in Distributed Systems," *IEEE Computer*, vol. 25, n. 6, pp. 8-17, 1992.

[59] Wang, H., Lin, T., Chen, C.H., Shen, Y., "Dynamic search in peer-to-peer networks," *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, 2004.

[60] "W3C," *http://www.w3.org/Protocols/*, acesso em julho de 2006.

[61] Yeager, W., Williams, J., "Secure Peer-to-Peer Networking: The JXTA Example," *IEEE IT Professional*, vol.4, n.2, pp.53-57, 2002.